Lucas Pereira da Silva

REUSO DE CÓDIGO E DE EXECUÇÃO DE *TEST FIXTURES* ENTRE CLASSES DE TESTE

Dissertação submetida ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Mestre em Ciências da Computação.

Orientadora: Profª. Drª. Patrícia Vilain

Florianópolis
2016

Ficha de identificação da obra elaborada pelo autor através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Silva, Lucas Pereira da
Reuso de código e de execução de test fixtures entre classes de teste / Lucas Pereira da Silva ; orientadora, Patrícia Vilain - Florianópolis, SC, 2016.
120 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico. Programa de Pós-Graduação em Ciência da Computação.

Inclui referências

1. Ciência da Computação. 2. Teste. 3. Test Fixture. 4. Fixture Setup. 5. Dependência de Teste. I. Vilain, Patrícia. II. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Ciência da Computação. III. Título.

Lucas Pereira da Silva

Reuso de Código e de Execução de *Test Fixtures* entre Classes de Teste

Esta dissertação foi julgada adequada para obtenção do título de mestre e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Florianópolis, 22 de julho de 2016.

_____
Profª. Carina Friedrich Dorneles, Drª.
Coordenadora do Programa

_____
Profª. Patrícia Vilain, Drª.
Universidade Federal de Santa Catarina
Orientadora

Banca Examinadora:

_____
Profª. Juliana Silva Herbert, Drª.
Universidade Federal de Ciências da Saúde de Porto Alegre
(Videoconferência)

_____
Prof. Raul Sidnei Wazlawick, Dr.
Universidade Federal de Santa Catarina

_____
Prof. Ricardo Pereira e Silva, Dr.
Universidade Federal de Santa Catarina

AGRADECIMENTOS

Agradeço fundamentalmente à minha orientadora Patrícia Vilain.

Aos membros da banca Juliana Silva Herbert, Raul Wazlawick e Ricardo Pereira e Silva.

Aos meus professores e professoras.

Aos meus amigos e amigas.

Aos meus pais.

Ao meu irmão.

Ao programa de pós-graduação em Ciência da Computação da Universidade Federal de Santa Catarina.

RESUMO

Teste de software consiste em uma atividade importante no processo de desenvolvimento e manutenção de software. A visão sobre sua utilidade tem evoluído nos últimos anos. Teste de software não é mais visto como uma atividade iniciada apenas ao final da etapa de codificação. Sua utilização passou a estar presente durante todo o processo de desenvolvimento e manutenção do software.

Testes modelam cenários possíveis de uso do Sistema em Teste (*System Under Test* - SUT). Nesse sentido, é necessário que o teste coloque o SUT em um estado que represente o cenário que está sendo modelado, ou seja, em um estado de interesse para o teste. Esta tarefa é realizada através da configuração de *test fixtures*. Um *test fixture* representa qualquer elemento necessário para exercitar o SUT. A parte da lógica do teste em que os *test fixtures* são configurados é chamada de *fixture setup*. Promover o reuso de *test fixtures* é importante para criar testes com melhor manutenibilidade e, consequentemente, reduzir o esforço de desenvolvimento dos testes.

Este trabalho propõe um conjunto de anotações que complementa o JUnit, para viabilizar tanto o reuso de código de *test fixtures* quanto o reuso de execução de *test fixtures*. A partir dessas anotações foi produzido o *framework* de teste Story. Experimentos mostram que através do Story foi possível atingir uma redução de 47,62% das linhas de código de teste e uma redução de 8 vezes no tempo de execução dos testes.

Palavras-chave: teste; *test fixture*; *fixture setup*; dependência de teste; reuso de código; reuso de execução; *framework* de teste.

ABSTRACT

Software Testing is an important activity of the software development process that has evolved in recent years. Software testing is no longer an activity that only starts after the coding phase. It is now carried out during the entire development process. In that sense, software testing incorporates other purposes, such as to identify failures, to prevent bug inclusion, to provide evidence that the software works, to give feedback about design and to be a way to specify and document the design.

A test case simulates a use scenario of the System Under Test (SUT). In this sense, a test case has to put the SUT in a state that represents the modeled scenario, which is, a state of interest to the tests. This is done through the execution of test fixtures. A test fixture represents everything that is necessary to exercise the SUT. Promoting test fixture reuse is important in order to create tests with a better maintainability and, as consequence, to reduce the effort of the test development.

This work proposes a set of annotations that complements JUnit in order to promote both the reuse of test fixture code and the reuse of test fixture execution. Through the annotations was implemented a new test framework called Story. Experiments show that Story achieved a reduction of 47,62% in the fixture setup code and approximately 8 times of execution time.

Keywords: testing; test fixture; fixture setup; test dependencies; test code reuse; test execution reuse; test framework.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Teste de software é uma atividade do processo de desenvolvimento e manutenção de software. A visão sobre sua utilidade tem evoluído nos últimos anos. Teste de software não é mais visto como uma atividade que é realizada apenas ao final da etapa de codificação para detectar falhas. A atividade de teste passou a estar presente durante todo o processo de desenvolvimento e manutenção de software (Bourque e Fairley, 2014). Nesse sentido, teste de software passou a ter propósitos variados, como: detectar falhas (Tiwari e Goel, 2013), prevenir que erros sejam introduzidos (Beizer, 1990), prover um indicativo de que o software funciona (Bertolino, 2007), auxiliar na compreensão do projeto e do código do sistema (Freeman e Pryce, 2009) e servir como um meio para especificação (Alvestad, 2007) e documentação dos requisitos (Haugset e Hanssen, 2008).

Segundo Tsai *et al.* (2003) a atividade de teste pode facilmente tomar de 50% a 60% de todo o esforço de desenvolvimento e manutenção de software. Para Meszaros (2006), a atividade de teste não deve aumentar o esforço total de desenvolvimento e manutenção do software, mesmo que exista garantia de melhoria da qualidade do software. O esforço gasto com a atividade de teste deve ser compensado por uma redução no esforço total de desenvolvimento e manutenção do software. Um dos fatores que contribui para que a atividade de teste tenha um custo benefício positivo é a automação de teste. A automação de teste consiste em criar, através de *scripts* de teste, testes automatizados que poderão ser executados durante todo o processo de desenvolvimento e manutenção do software. Segundo Meszaros (2006), o esforço para construir e manter testes automatizados deve ser justificado pelos benefícios provenientes da atividade de teste. Para isso, defende-se que sejam satisfeitas as seguintes características: (1) testes devem ser fáceis de executar, isto é, completamente automatizados, auto verificáveis e repetíveis; (2) testes devem ser fáceis de ler e escrever, isto é, simples, expressivos e com separação de conceitos; e (3) testes devem ser robustos, isto é, requerer o mínimo de manutenção conforme o software evoluí.

Testes devem ser completamente automatizados para que possam ser executados sem nenhum esforço adicional. Testes devem ser auto verificáveis para que possam detectar e reportar erros sem necessidade de intervenção manual. Testes devem ser repetíveis para que possam ser executados múltiplas vezes produzindo sempre o mesmo resultado. Testes devem ser simples para que contenham o mínimo de detalhes

necessário. Testes devem ser expressivos para que possam comunicar a intenção daquilo que está sendo testado. Testes devem ter separação de conceitos para que a lógica do teste se mantenha separada da lógica do código de produção e para que cada teste possa se preocupar com um único conceito. Testes devem ser robustos para que alterações no software não afetem de forma desproporcional os testes já existentes, ou seja, espera-se que pequenas mudanças na implementação do software afetem no máximo um pequeno conjunto de testes. Por exemplo, se uma simples mudança na assinatura de um método afetar uma grande quantidade de testes, então as alterações necessárias nos testes serão desproporcionais à alteração realizada na implementação do software.

Meszaros (2006) vê a atividade de teste como parte intrínseca do processo de desenvolvimento de software. Testes automatizados devem, portanto, ser incorporados ao processo de desenvolvimento de modo que a evolução do software seja acompanhada pela evolução dos testes. Quanto mais cedo um teste for adicionado mais cedo serão percebidos os seus potenciais benefícios. Por outro lado, maior será a necessidade de garantir a satisfação das características previamente mencionadas. Isso se deve ao fato de que quanto mais cedo um teste for adicionado mais vezes o teste será lido e executado e maiores serão as chances de que o teste seja afetado por futuras modificações no software.

Existe, portanto, um impasse quanto à incorporação da atividade de teste já nas fases iniciais de desenvolvimento. Por um lado, isto é importante para que os potenciais benefícios dos testes possam ser percebidos ao longo de todo o desenvolvimento e manutenção do software. Por outro lado, isto traz uma responsabilidade extra quanto à manutenibilidade do código de teste.

De acordo com Berner, Weber e Keller (2005), a manutenção dos testes tende a ter um impacto muito maior no esforço total do projeto do que a implementação inicial dos testes. Segundo Greiler *et al.* (2013), assim como o código de produção, o código de teste também precisa ser mantido, compreendido e ajustado. O sucesso a longo prazo da automação de testes é fortemente influenciado pela manutenibilidade do código de teste. Para que possa existir uma boa manutenibilidade, os métodos de teste precisam ser estruturados claramente, ter nomes representativos e ter pouco código. Além disso, a duplicação de código entre os métodos de teste deve ser evitada. Esses princípios estão alinhados com as características que Meszaros (2006) defende para os testes.

Segundo Emery (2009), o motivo comum pelo qual pessoas e organizações abandonam a automação de testes é que os testes se

tornam frágeis e com alto custo de manutenção. Pequenas mudanças na implementação do software podem afetar uma grande quantidade de testes, e consertá-los demanda grande esforço. Não se pode impedir que os requisitos e a implementação do software mudem. Por isso, a melhor forma para manter os custos de manutenção dos testes baixos é tonar os testes mais adaptáveis a esse tipo de mudança. Para o autor, um dos fatores chaves para isso é evitar duplicações entre os testes.

Testes modelam cenários possíveis de uso do Sistema em Teste (*System Under Test* - SUT). Nesse sentido, é necessário que o teste coloque o SUT em um estado que represente o cenário que está sendo modelado, ou seja, em um estado de interesse para o teste (Greiler, Deursen e Storey, 2013). Esta tarefa é realizada através de configurações de *test fixtures*. De acordo com Meszaros (2006), *test fixture* representa qualquer elemento necessário para exercitar o SUT. A parte da lógica do teste em que os *test fixtures* são configurados é chamada de *fixture setup*.

De acordo com Berner, Weber e Keller (2005), testes tendem a ser repetitivos e apresentam um alto potencial de reuso. Conforme o software evolui e o número de testes aumenta, é comum que comecem a aparecer *test fixtures* duplicados através de diferentes *fixtures setups*. O gráfico apresentado na Figura 1 (Berner, Weber e Keller, 2005) compara o desenvolvimento de casos de teste com e sem reuso de *test fixtures*. O gráfico apresentado na Figura 1 foi gerado a partir de um software do departamento de vendas de uma grande companhia internacional do setor industrial. A metodologia XP (*Extreme Programming*) (Beck, 2000) foi aplicada durante o desenvolvimento do referido software. No caso do desenvolvimento com reuso de *test fixtures*, percebe-se uma redução no esforço de desenvolvimento dos testes conforme o número de casos de teste implementados aumenta. Por outro lado, quando os *test fixtures* não são reusados, o esforço se mantém praticamente constante durante a maior parte do tempo e sofre um pequeno aumento no final. Portanto, aumentar o reuso do código de teste contribui para reduzir o esforço de desenvolvimento dos testes.

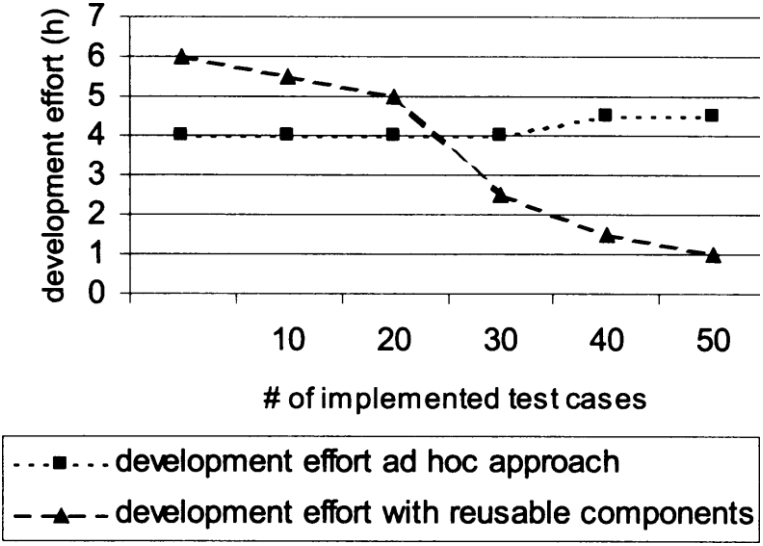


Figura 1. Esforço de desenvolvimento de testes com e sem reuso.

Existem diferentes maneiras para configurar *test fixtures*. Meszaros (2006) apresenta estratégias de *fixture setup* que podem ou não promover o reuso de *test fixture*. Dentre as estratégias apresentadas destacam-se, *inline setup*, *implicit setup* e *delegate setup*. Os testes da Figura 1 que foram implementados sem reuso de *test fixtures* utilizam a estratégia *inline setup*. Nesta estratégia cada teste contém todo o código necessário para configurar os *test fixtures* – o reuso de código não é promovido mesmo que testes diferentes tenham *test fixtures* iguais. A grande vantagem da estratégia é facilitar a compreensão da relação de causa e efeito entre os *test fixtures* e as saídas do SUT, uma vez que os *test fixtures* são configurados dentro do próprio método de teste ficando próximos, portanto, das verificações do teste. Entretanto, sua grande desvantagem é a duplicação do código de teste.

A Figura 2 apresenta dois testes onde a estratégia *inline setup* é utilizada. Pode-se observar que existe duplicação de código entre os métodos `createUser` e `insertUser`. Esse tipo de duplicação afeta a manutenibilidade dos testes a longo prazo. Se, por exemplo, for feita uma modificação na classe `User`, de modo que os atributos da classe passem a ser recebidos através do construtor e não mais através dos métodos *setters*, então os dois testes apresentados na Figura 2 serão afetados pela mudança.

```
public class UserTest {

  @Test
  public void createUser() {
    User john = new User();
    john.setName("John");
    john.setCareer("Teacher");
    assertNull(john.id());
    assertEquals("John", john.name());
    assertEquals("Teacher", john.career());
  }

  @Test
  public void insertUser() {
    User john = new User();
    john.setName("John");
    john.setCareer("Teacher");
    UserDao userDao = new UserDao();
    Integer id = userDao.insert(john);
    assertNotNull(id);
    assertEquals(id, john.id());
  }

}
```

Figura 2. Testes com duplicação de *test fixture*.

Os testes da Figura 1 que foram implementados com reuso de *test fixtures* utilizam uma combinação das estratégias *inline setup* e *delegate setup*. A estratégia *delegate setup* utiliza métodos auxiliares onde os *test fixtures* são configurados. Em geral, estes métodos auxiliares são colocados em classes separadas das classes de teste. Os métodos auxiliares podem, então, ser chamados por qualquer teste, promovendo, dessa forma, o reuso de *test fixtures*.

Cada estratégia de *fixture setup* possui diferentes características que impactam no código de teste. Com exceção da estratégia *inline setup*, as demais estratégias apresentadas por Meszaros (2006) contribuem, em algum nível, com o reuso de código de *test fixture*. Entretanto, não existe uma estratégia ideal, mas sim estratégias com características diferentes que se adequam melhor a diferentes situações. Por exemplo, a estratégia *implicit setup* é adequada para os casos onde testes da mesma classe necessitam dos mesmos *test fixtures*. Nesta estratégia os testes de uma mesma classe compartilham o código de configuração de *test fixtures* disponibilizado através um método especial da classe, o *setup method*. Já a estratégia *delegate setup* é adequada para encapsular *test fixtures* complexos através de métodos auxiliares. Nenhuma das estratégias pode resolver completamente o problema de

duplicação de *test fixtures*. Algumas estratégias possibilitam um maior reuso de *test fixtures* do que outras, mas é importante que se perceba que outras características também devem ser levadas em consideração no momento de se escolher a estratégia de *fixture setup*. Em alguns casos pode ser mais adequado optar pela estratégia *inline setup* que prioriza a compreensão do código de teste em detrimento, por exemplo, da estratégia *delegate setup* que prioriza o reuso de *test fixtures*.

O problema que este trabalho pretende resolver é permitir que classes de teste utilizem *test fixtures* definidos em uma ou mais classes de teste já existentes sem que, para isso, seja necessário alterar a estrutura das classes envolvidas e sem causar dependência entre testes. Nenhuma das estratégias de *fixture setup* apresentadas por Meszaros (2006) permite isso.

1.1. HIPÓTESE DE PESQUISA

A hipótese de pesquisa deste trabalho é que *test fixtures* definidos em uma classe podem ser reutilizados por testes de outras classes sem que haja dependência entre os testes.

1.2. OBJETIVOS

O objetivo geral deste trabalho consiste na definição de modelos que permitam a implementação de uma estratégia de *fixture setup* que promova o aumento do reuso de *test fixtures* e contribua para o desenvolvimento de testes com boa manutenibilidade.

Para alcançar o objetivo geral, os seguintes objetivos específicos deverão ser alcançados:

- Promover o reuso de código de *test fixture*.
- Promover o reuso de execução de *test fixture*.
- Implementar os modelos propostos como uma estratégia de *fixture setup* de um *framework* de teste.
- Avaliar comparativamente as estratégias de *fixture setup* existentes e a estratégia proposta.

1.3. ORGANIZAÇÃO DESTE TRABALHO

Os capítulos deste trabalho são organizados da seguinte forma: o Capítulo 2 apresenta conceitos básicos sobre o desenvolvimento de testes; o Capítulo 3 apresenta os trabalhos relacionados com este

trabalho; o Capítulo 4 apresenta a proposta deste trabalho através de duas perspectivas: reuso de código e reuso de execução; o Capítulo 5 apresenta o Story, um *framework* de teste onde a proposta deste trabalho foi implementada; o Capítulo 6 apresenta, através de um exemplo de uso e de experimentos, a avaliação da proposta deste trabalho; e, por fim, o Capítulo 7 apresenta as conclusões e trabalhos futuros.

2. FUNDAMENTAÇÃO

Um **teste** é um procedimento, manual ou automático, que pode ser usado para verificar se um dado **SUT** funciona de acordo com o comportamento esperado. O SUT é o sistema em teste, e representa a parte do software que está sendo testada. Cada teste deve realizar uma série de configurações para que possa colocar o SUT no estado desejado. Essas configurações recebem o nome de **test fixtures**. A parte da lógica do teste onde os *test fixtures* são configurados é chamada de **fixture setup.** Após realizar as configurações, o teste realiza uma série de **verificações** para garantir que o SUT funcionou de acordo com o comportamento esperado.

Os exemplos apresentados nesse trabalho são baseados no *framework* de teste JUnit[1], desenvolvido para a linguagem Java. A Figura 3 apresenta um exemplo de teste escrito através do JUnit. A anotação `@Test` indica ao *framework* que o método anotado corresponde a um teste. Essa abordagem segue a definição utilizada por Freeman e Pryce (2009), onde, tipicamente, cada teste é separado em um método de teste.

```
public class TesteBanco {

  @Test
  public void umBanco() {
    SistemaBancario sistemaBancario = new SistemaBancario();
    Banco bancoDoBrasil = sistemaBancario.criarBanco("Banco do Brasil", Moeda.BRL);
    assertEquals("Banco do Brasil", bancoDoBrasil.obterNome());
    assertEquals(Moeda.BRL, bancoDoBrasil.obterMoeda());
  }

}
```

Figura 3. Teste com a estratégia *inline setup*.

Na Figura 3, o teste corresponde ao método `umBanco`. As duas primeiras linhas do método correspondem ao *fixture setup* do teste, e é nestas linhas que estão definidos os *test fixtures*. O objeto `sistemaBancario`, o objeto `bancoDoBrasil`, o texto "`Banco do Brasil`", e o valor `Moeda.BRL` são todos exemplos de *test fixture*. As duas últimas linhas do método de teste correspondem às verificações do teste. O SUT não aparece representado na figura, mas corresponde a toda a lógica que é executada nas classes `Banco` e `SistemaBancario` durante a execução do teste.

[1] http://junit.org/

2.1. FIXTURE SETUP E TEST FIXTURE

O *fixture setup* compreende da parte da lógica do teste onde *test fixtures* são configurados. Cada teste possui um *fixture setup*, que consiste da definição de todos *test fixtures* que serão utilizados no teste. O *fixture setup* de um teste pode ser definido por diferentes estratégias de *fixture setup*.

Um dos problemas que afetam a manutenibilidade do código de teste é a duplicação de *test fixtures*. Nem sempre é simples e/ou possível evitar esse tipo de problema, porém existem diferentes estratégias de *fixture setup* que podem ser utilizadas para reduzir a duplicação de código. Cada estratégia se adequa melhor a tipos diferentes de situações.

2.1.1. Propriedades dos *Test Fixtures*

Para facilitar a discussão a respeito das diferentes estratégias de *fixture setup*, iremos apresentar algumas propriedades aplicadas a *test fixtures*. Algumas dessas propriedades foram extraídas da literatura, enquanto que outras foram definidas neste trabalho.

2.1.1.1. Transiente

Segundo Meszaros (2006), *test fixtures* transientes são aqueles que são automaticamente removidos do SUT assim que a execução do teste é encerrada. Em geral são objetos na memória. O desenvolvedor do teste não precisa se preocupar em limpar o SUT para que este seja mantido em um estado consistente para o próximo teste.

2.1.1.2. Persistente

Ao contrário do *test fixture* transiente, o persistente é aquele que permanece no SUT mesmo depois que a execução do teste é encerrada (Meszaros, 2006). Um *test fixture* persistente pode ser um registro do banco de dados, um arquivo ou até mesmo um atributo estático. O desenvolvedor do teste deve ter cautela com *test fixtures* persistentes, pois podem deixar o SUT em um estado inconsistente para o próximo teste.

2.1.1.3. Particular

Um *test fixture* particular é aquele que é utilizado por apenas uma execução de teste. Um *test fixture* particular não pode ser utilizado em diferentes execuções de teste, mesmo se forem execuções distintas do mesmo teste.

2.1.1.4. Coletivo

Um *test fixture* coletivo é aquele que é utilizado por mais de uma execução de teste. Um *test fixture* coletivo pode ser utilizado em diferentes execuções de teste, inclusive se forem execuções de testes distintos.

2.1.1.5. Manuseável

Um *test fixture* é dito manuseável por um dado teste quando o teste pode ter acesso ao *test fixture* através de algum componente concreto do código de teste, como, por exemplo, através de um atributo da classe, de uma variável local ou até mesmo um retorno de um método. Essa propriedade deve ser vista sempre da perspectiva do teste. Por exemplo, um *test fixture* pode ser manuseável para um determinado teste e não ser para outro.

2.2. ESTRATÉGIAS DE FIXTURE SETUP

Meszaros (2006), apresenta diversas estratégias de *fixture setup*. As estratégias apresentadas são divididas em duas categorias: *fresh fixture setup* e *shared fixture construction*. Em ambas categorias existem estratégias que possibilitam o reuso de código. Entretanto, apenas na categoria *shared fixture construction* é possível o reuso de execução.

2.2.1. Fresh Fixture Setup

A categoria *fresh fixture setup* é composta por três estratégias, sendo elas: *inline setup*, *implicit setup* e *delegate setup*. Nesta categoria os *test fixtures* são sempre particulares. Portanto, nesta categoria há apenas a possibilidade de reuso de código, mas não reuso de execução.

2.2.1.1. Inline Setup

Na estratégia *inline setup* os *text fixtures* são configurados dentro do próprio método de teste. Esta estratégia configura *test fixtures* particulares e manuseáveis apenas pelo próprio teste. Em geral, esta estratégia é utilizada em testes que necessitam de *test fixtures* muito específicos ou no desenvolvimento dos primeiros testes onde ainda não existe a necessidade de reuso de *test fixtures*.

A Figura 3, apresentada anteriormente, mostra um exemplo de teste em a estratégia *inline setup* é utilizada. As duas primeiras linhas do teste contêm a configuração dos *test fixtures* para o teste. Destaca-se que na estratégia *inline setup* a configuração dos *test fixtures* é realizada diretamente no método de teste.

Como os *test fixtures* são configurados diretamente no método de teste, torna-se fácil compreender a relação de causa e efeito entre os *test fixtures* e as saídas do SUT. Entretanto, a longo prazo, esta estratégia pode levar a considerável duplicação de código, uma vez que diversos testes podem utilizar *test fixtures* idênticos ou muito similares. Além disso, esta estratégia pode dificultar a compreensão do teste nos casos em que os *test fixtures* forem muito complexos.

2.2.1.2. Implicit Setup

Na estratégia *implicit setup* os *test fixtures* são configurados através de um método especial, comumente denominado de *setup method*, pertencente à classe de teste. Esta estratégia configura *test fixtures* particulares e potencialmente manuseáveis pelos métodos da classe onde o *setup method* foi definido.

A Figura 4 apresenta um exemplo da estratégia *implicit setup*. No exemplo o método `configurar` representa o *setup method*. O atributo `bancoDoBrasil` é utilizado para permitir que o *test fixture* seja manuseável pelos testes da classe. Quando um teste da classe é executado, o *framework* fica responsável por descobrir e executar o *setup method*. No JUnit a anotação `@Before` é utilizada para indicar o *setup method*.

```
public class TesteBanco {

  private Banco bancoDoBrasil;

  @Test
  public void configurar() {
    SistemaBancario sistemaBancario = new SistemaBancario();
    bancoDoBrasil = sistemaBancario.criarBanco("Banco do Brasil", Moeda.BRL);
  }

  @Test
  public void umBanco() {
    assertEquals("Banco do Brasil", bancoDoBrasil.obterNome());
    assertEquals(Moeda.BRL, bancoDoBrasil.obterMoeda());
  }

}
```

Figura 4. Teste com a estratégia *implicit setup*.

A vantagem da estratégia é promover o reuso de *test fixtures* entre testes de uma classe. Porém, para utilizar esta estratégia, os testes devem ser agrupados na mesma classe. Isso pode trazer uma complicação a longo prazo, pois torna-se cada vez mais difícil agrupar testes que necessitem exatamente dos mesmos *test fixtures*. Com facilidade alguns testes irão necessitar de *test fixtures* específicos.

Greiler *et al.* (2013) apresenta fedores[2] que podem dificultar a manutenibilidade do código de teste quando esta estratégia é utilizada. Um dos principais problemas apontados ocorre quando existem *test fixtures* que são necessários apenas para alguns testes da classe. Isso pode prejudicar a legibilidade dos demais testes, uma vez que existirá mais *test fixtures* do que apenas os necessários.

Existem casos onde classes de teste distintas apresentam *setup methods* idênticos. Uma possível solução para remover a duplicação de código nesses casos consistiria em juntar as duas classes de teste em uma só. Assim, o mesmo *setup methods* seria utilizado pelas duas classes. Entretanto, essa opção interfere na liberdade de organização das classes de testes.

2.2.1.3. Delegate Setup

A estratégia *delegate setup* utiliza métodos auxiliares onde os *test fixtures* são configurados. Em geral, esses métodos são disponibilizados através de classes auxiliares, separadas das classes de teste. Esta estratégia realiza a configuração de *test fixtures* particulares e potencialmente manuseáveis por qualquer teste. É importante destacar que apenas o *text fixture* retornado pelo método auxiliar será

[2] Do termo inglês, *bad smell*.

manuseável. Por isso, tipicamente, esta estratégia é utilizada quando se deseja configurar um *test fixture* complexo, de modo que detalhes irrelevantes fiquem escondidos atrás de um método nomeado adequadamente.

A Figura 5 mostra um exemplo da estratégia *delegate setup*. O *test fixture* necessário para o teste é configurado através do método `criarBancoDoBrasil` da classe `Auxiliar`. O método auxiliar é chamado dentro do próprio método de teste. O retorno do método auxiliar é, então, atribuído à variável `bancoDoBrasil`.

```
public class TesteBanco {

  @Test
  public void umBanco() {
    Banco bancoDoBrasil = Auxiliar.criarBancoDoBrasil();
    assertEquals("Banco do Brasil", bancoDoBrasil.obterNome());
    assertEquals(Moeda.BRL, bancoDoBrasil.obterMoeda());
  }

}

public class Auxiliar {

  public static Banco criarBancoDoBrasil() {
    SistemaBancario sistemaBancario = new SistemaBancario();
    return sistemaBancario.criarBanco("Banco do Brasil", Moeda.BRL);
  }

}
```

Figura 5. Teste com a estratégia *delegate setup*.

A principal vantagem desta estratégia é permitir que o mesmo código de teste possa ser utilizado por testes diferentes de classes diferentes. A desvantagem da estratégia é que em alguns casos, a utilização da estratégia pode dificultar a compreensão da relação de causa e efeito entre os *test fixtures* e as saídas do SUT, uma vez que o método auxiliar é, comumente, colocado em uma classe separada do teste.

2.2.2. Shared Fixture Construction

As estratégias da categoria *shared fixture construction* realizam a configuração de *test fixtures* coletivos. Dessa forma, um *test fixture* pode ser criado uma única vez e reutilizado entre várias execuções de teste distintas. Por isso, as estratégias devem gerenciar os *test fixtures* para que possam ser usados em diferentes execuções de teste. Cada estratégia da categoria possui um modo para realizar esse gerenciamento. Por exemplo, a estratégia pode manter o *test fixture*

salvo no sistema de arquivos, em um banco de dados, através de um atributo estático ou até mesmo utilizar algum mecanismo interno do *framework* de teste. O momento em que os *test fixtures* são configurados varia de acordo com cada estratégia da categoria.

Os *frameworks* de teste atuais não apresentam formas nativas e nem padronizadas para realizar este tipo de *fixture setup*. Por isso, é necessário que o desenvolvedor do teste assuma esse papel e passe a gerenciar parte do fluxo de execução dos testes, ferindo, assim, o princípio de *Hollywood*[3] (Sweet, 1985; Sobernig e Zdun, 2010). Ao assumir o fluxo de execução, o desenvolvedor pode afetar o estado interno do *framework* fazendo com que os testes falhem devido a um gerenciamento incorreto (Mattsson, Bosch e Fayad, 1999).

Apesar das estratégias desta categoria promoverem, além do reuso de código, o reuso de execução, é necessário que o desenvolvedor assuma o controle de execução do *framework* para garantir que os testes recebam os *test fixtures* adequados. Esse controle extra demanda um maior esforço do desenvolvedor durante o desenvolvimento do código de teste.

A principal vantagem das estratégias desta categoria é a configuração de *test fixtures* coletivos. O benefício em utilizar *test fixtures* coletivos está em promover reuso de execução. O reuso de execução ocorre, pois um mesmo *test fixture* é utilizado em diferentes execuções de teste. Porém, é necessário ter cautela ao manusear *test fixtures* coletivos. Um teste pode manusear incorretamente um *test fixture* coletivo, de modo que isso deixe o SUT em um estado inconsistente para um próximo teste.

2.3. DISCUSSÃO

As estratégias *implicit setup* e *delegate setup* possibilitam o reuso de código de *test fixture*, entretanto apresentam limitações. Na estratégia *implicit setup* é possível promover o reuso apenas entre testes de uma mesma classe. Já na estratégia *delegate setup* é possível promover o reuso entre testes de classes diferentes, entretanto cada método auxiliar do *delegate setup* pode ter apenas um *test fixture* manuseável.

As estratégias da categoria *shared fixture construction* possibilitam tanto reuso de código quanto reuso de execução de *test fixture*. O principal problema com as estratégias dessa categoria é que os

[3] O princípio de Hollywood defende que é o *framework* quem deve chamar a aplicação, e não o contrário – *"Don't call us, we'll call you"*.

test fixtures são coletivizados entre os testes de forma não padronizada. Isso cria uma dependência entre os testes, pois se um teste modificar um *test fixture* coletivo, isto irá afetar a execução de um próximo teste que utilizar o mesmo *test fixture*. Além disso, como os *frameworks* de teste não incorporam de forma nativa as estratégias desta categoria é necessário que se viole o princípio de Hollywood e isso aumenta o esforço necessário para se desenvolver os testes.

As estratégias de *fixture setup* apresentadas neste capítulo não permitem que testes utilizem *test fixtures* provenientes de outras classes de teste sem que seja criada uma relação de dependência entre os testes. Além disso, o reuso de execução de *test fixture* somente é possível se o desenvolvedor do teste assumir a responsabilidade de gerenciar os *test fixtures* coletivos. Não foi encontrada, portanto, uma estratégia que permita promover o reuso de código de *test fixture* entre classes de teste e que, ao mesmo tempo, possibilite que o próprio *framework* de teste promova automaticamente o reuso de execução de *test fixture* nos casos em que isto for possível.

3. TRABALHOS RELACIONADOS

Este capítulo apresenta os trabalhos relacionados mais relevantes para este trabalho. Considerou-se os trabalhos que abordam o reuso de *test fixtures*. Para realizar a busca por trabalhos correlatos foram utilizadas as bibliotecas digitais da IEEE, ACM e Springer. Não foi realizada nenhuma revisão sistemática. A busca foi realizada de forma *ad hoc* através das bases mencionadas. Além disso, as referências mais relevantes presentes nos trabalhos encontrados através destas bibliotecas digitais também foram consultadas. Os três trabalhos relacionados mais relevantes para esta pesquisa são apresentados a seguir.

3.1. CHRISTENSEN ET AL.

Christensen *et al.* (2006) apresenta uma extensão aos *frameworks* de teste que possibilita reutilizar *test fixtures* em testes que envolvem banco de dados. Na extensão proposta são declaradas dependências entre classes de teste. Essas dependências devem refletir a mesma dependência existente entre entidades do banco de dados. Assim, os testes das dependências de uma determinada classe são executados de forma recursiva antes da execução dos testes da própria classe.

O trabalho considera que deverá ser definida uma classe de teste para cada tabela do banco de dados que será testada. Na abordagem adotada é necessário que também sejam definidos testes para as tabelas que forem chaves estrangeiras de tabelas testadas. Dessa forma, antes de executar um teste onde é realizada a inserção de um dado, primeiro será executado um teste onde o dado que corresponde à chave estrangeira é inserido.

A aplicação de dependências entre as classes de teste de um projeto de tamanho médio mostrou que houve uma redução de 40% do tempo de execução dos testes e uma redução de 25% de linhas de código de teste. A redução no tempo de execução acontece, porque os *test fixtures* persistidos através do banco de dados são coletivos, ou seja, são usados em vários testes sem precisarem ser armazenados no banco de dados a cada execução do teste.

Na proposta para cada teste que realiza a inserção de um dado deverá ser definido um teste para remover o dado inserido. Essa abordagem foi adotada para que o banco de dados seja sempre mantido em um estado consistente. Por exemplo, se houver uma restrição de duplicação definida no banco de dados e se um teste de inserção for executado duas vezes consecutivas sem que o banco de dados tenha sido

limpo entre as execuções, então irá ocorrer uma falha na segunda vez em que o teste for executado. Para resolver esse problema, o trabalho obriga que para cada teste que realiza a inserção de um dado no banco deva existir um teste que realiza a remoção do respectivo dado. Assim, o *framework* ordena a execução dos testes de tal forma que primeiro são executados os testes de inserção e por último são executados os testes de remoção. Com isso, ao terminar a execução dos testes, o banco de dados sempre é deixado no mesmo estado em que estava quando a execução dos testes foi iniciada.

Uma limitação do trabalho é que os *test fixtures* são manuseáveis apenas pelos testes da própria classe. *Test fixtures* de outras classes não poder ser manuseáveis nas classes dependentes.

A proposta cria encadeamentos de classes de teste dependentes. Assim, são criadas sequências de teste. Cada teste de uma sequência poderá utilizar os *test fixtures* coletivos de testes anteriores. O problema dessa abordagem é que nenhum teste pode alterar os *test fixtures* coletivos, pois caso o faça, poderá deixar o SUT em um estado inconsistente para o próximo teste.

3.2. MUGRIDGE E CUNNINGHAM

Mugridge e Cunningham (2005) abordam o reuso de *test fixtures* na perspectiva do desenvolvimento dirigido a histórias. Os autores analisam histórias de teste criadas através do Fit. O Fit é um *framework* para a criação de testes de aceitação através de uma especificação em formato tabular (Borg e Kropp, 2011). Após criadas as especificações, o desenvolvedor do teste deve, então, criar o código para automatizar os testes. O fluxo de execução de uma história de teste pode ser dividido em três partes: (1) configuração dos *test fixtures*; (2) exercício do SUT; e (3) verificação das saídas do SUT.

O trabalho chama atenção para o fato de que em um grande número de histórias de teste existe uma considerável quantidade de duplicação das tabelas Fit que representam os *test fixtures*. Uma percepção importante é a de que algumas tabelas usadas na etapa inicial de uma determinada história são iguais àquelas usadas na etapa final de uma outra história de teste.

Os autores propõem uma mudança na forma em que as histórias de teste são criadas. Propõe-se que primeiro sejam especificadas as tabelas que representam os *test fixtures*. Após isso, as histórias de teste deverão ser criadas através da composição das tabelas previamente definidas. O passo seguinte consiste da criação de um grafo onde as

diversas histórias de teste são conectadas entre si através das tabelas reusadas. O conjunto das histórias de teste passa a ser visto, então, como um grande grafo de teste que poderá ser executado. Entretanto, a execução de um grafo de teste, ao invés da execução de histórias de teste, torna mais complexo o rastreamento de erros nos casos onde os testes falham.

Histórias de teste permitem facilitar a compreensão do próprio teste e do software que está sendo testado. Na proposta apresentada essa característica torna-se menos efetiva, pois, mesmo que as histórias ainda sejam criadas individualmente, é necessário que primeiro sejam definidas as tabelas que serão reusadas.

3.3. LONGO ET AL.

O trabalho proposto por Longo *et al.* (2015) utiliza uma linguagem de notação de objetos baseada em JSON para descrever *test fixtures*. Os *test fixtures* são configurados através de arquivos com a extensão `picon` que ficam em um local separado dos testes. O objetivo do trabalho é criar um repositório centralizado de *test fixtures* que poderão ser utilizados por qualquer teste. Cada *test fixture* deve ser identificado através de um qualificador. Os testes podem manusear os *test fixtures* através da declaração de um atributo na classe de teste. O atributo deve ser nomeado de acordo com o qualificador do *test fixture* desejado.

No trabalho é apresentada a ferramenta Picon. Esta ferramenta é integrada ao *framework* de teste JUnit. Quando um teste do JUnit é executado, a ferramenta procura por *test fixtures* que correspondem a atributos da classe de teste. Os *test fixtures* necessários para o teste são injetados nos atributos correspondentes. É importante destacar que os *test fixtures* são particulares. Isto significa que a cada execução de teste os *test fixtures* são configurados novamente.

Longo et al. defende que o sucesso da utilização da proposta é fortemente influenciado pelos qualificares utilizados. Como os *test fixtures* são configurados em um local separado do teste, é necessário que se utilizem qualificadores que sejam ao mesmo tempo expressivos e coesos. Essa característica é importante para que o desenvolvedor do teste possa identificar com facilidade cada *test fixture* apenas através de seu qualificador. Caso contrário, torna-se difícil compreender a relação de causa e efeito entre os *test fixtures* e as saídas do SUT.

Uma das limitações da abordagem é que, como a injeção de *test fixtures* é realizada dinamicamente, eventuais inconsistências entre os

qualificadores dos *test fixtures* e o nome dos atributos somente serão percebidos durante a execução dos testes. Outra limitação é que a configuração dos *test fixtures* deve ser descrita através de uma linguagem específica, diferente da linguagem em que o teste é escrito.

A proposta foi aplicada em um ambiente de desenvolvimento com TDD onde o número de linhas de código de teste foi medido através de um período total de 6000 horas de desenvolvimento. Estimou-se que a aplicação da proposta possibilitou uma redução de 60% de linhas de código de teste. Como a proposta utiliza *test fixtures* particulares, não houve redução no tempo de execução dos testes.

3.4. DISCUSSÃO

Todos os trabalhos apresentados abordam o reuso de *test fixtures*. Entretanto, cada trabalho é aplicado a um contexto diferente. O trabalho proposto por Christensen *et al.* (2006) é o que aborda o contexto mais parecido com o contexto deste trabalho. Na abordagem proposta, é promovido o reuso de código de *test fixture* entre classes de teste dependentes para testes envolvendo banco de dados.

O trabalho proposto por Mugridge e Cunningham (2005) aborda o reuso de *test fixtures* aplicado a um contexto bastante diferente da proposta deste trabalho. Os autores abordam o reuso de *test fixtures* para testes de aceitação com o Fit. Assim, é proposto o reuso de *test fixtures* definidos através de tabelas Fit. Tabelas Fit idênticas podem ser reusadas através de diversas histórias de teste.

Assim como neste trabalho, o trabalho proposto por Longo *et al.* (2015) também aborda o reuso de código de *test fixture*. A principal diferença é que na abordagem proposta por Longo *et al.* (2015) os *test fixtures* são centralizados em um repositório separado das classes de teste.

Neste trabalho é abordado o reuso tanto de código quanto de execução de *test fixture*. O objetivo é que testes possam reutilizar *test fixtures* provenientes de outras classes de teste sem que isso gere uma dependência entre os testes. Além disso, o mesmo mecanismo utilizado para promover o reuso de código de *test fixture* deve poder ser utilizado por *frameworks* de teste para promover de forma transparente o reuso de execução de *test fixtures*.

Não é possível realizar uma comparação direta entre os trabalhos relacionados e este trabalho, pois os contextos em que os trabalhos são aplicados diferem consistentemente. O denominador comum entre os trabalhos é o reuso de *test fixtures*, entretanto, em cada trabalho esse

reuso é aplicado através de um contexto de desenvolvimento de testes diferente. O contexto abordado neste trabalho abrange testes que sejam criados a partir do **desenvolvimento de código de teste**, podendo ser qualquer tipo de teste (desde testes de unidade até testes de aceitação) e para qualquer camada do software (desde a camada de persistência até a camada de interface gráfica com o usuário). Na Tabela 1 são apresentadas as principais características dos trabalhos relacionados e deste trabalho considerando um contexto genérico de desenvolvimento de testes.

	Christensen et al. (2006)	**Mugridge e Cunningham (2005)**	**Longo et al. (2015)**	**Proposta**
Reuso de código	Sim	Sim	Sim	Sim
Onde são definidos os *test fixtures* reusáveis	Classes de teste	Repositório de tabelas Fit	Repositório Picon	Classes de teste
Quem pode reutilizar os *test fixtures*	Testes de classes dependentes	Qualquer história de teste	Qualquer teste	Testes de classes dependentes
Test fixtures manuseáveis	Não	Não	Sim	Sim
Reuso de execução	Sim	Sim	Não	Sim
Garantia de independência entre testes	Não	Não	Sim	Sim

Tabela 1. Comparação com trabalhos relacionados.

4. PROPOSTA

Este trabalho aborda o reuso de *test fixtures* como medida para melhorar a manutenibilidade do código de teste através da redução de duplicação de código de teste. A duplicação de código é um problema inerente ao próprio processo de desenvolvimento de software, e não apenas ao desenvolvimento de testes. A Engenharia de Software busca constantemente estratégias para promover o reuso (não necessariamente de código) e, assim, facilitar o processo de desenvolvimento de software. Bibliotecas, *frameworks* e o paradigma de programação orientada a objetos são todas estratégias que contribuem para promover o reuso de código. Entretanto, o código de teste requer cuidados extras que vão além da questão do reuso. O código de teste deve ter um caráter descritivo, com uma lógica simples, para que a complexidade do código de teste não seja superior à complexidade do código de produção (Meszaros, 2006). Caso a complexidade para se compreender e desenvolver o código de teste seja maior que para o código de produção, então os testes perdem a eficácia, uma vez que as chances de existirem erros no código de teste serão tão grandes quanto, ou até mesmo maiores que, as chances de existirem erros no código de produção.

Existem diferentes estratégias de *fixture setup*. Cada estratégia tem impactos distintos no reuso de *test fixtures* e na complexidade do código de teste. Por isso, não basta escolher a estratégia que melhor promova o reuso de *test fixtures*. A estratégia adotada deve manter uma harmonia entre ambos, o reuso de *test fixtures* e a complexidade do código de teste. A estratégia de *fixture setup* que melhor mantém essa harmonia desejada varia conforme o contexto em que os *test fixtures* são necessários. É por esse motivo que existem estratégias diferentes. Não existe uma estratégia absoluta, mas sim estratégias que se adequam melhor a diferentes tipos de contextos. Meszaros (2006) defende que um bom código de teste é aquele em que as diferentes estratégias de *fixture setup* são combinadas de modo que o reuso de *test fixtures* seja maximizado e a complexidade do código de teste minimizada.

Deseja-se com esse trabalho promover o reuso de *test fixtures* em contextos em que as estratégias de *fixture setup* existentes não são tão adequadas. Esta proposta pretende contribuir na configuração do contexto onde os testes são construídos de forma que reflitam uma sequência cronológica de utilização do SUT. Um contexto é definido quando um teste tem como ponto de partida um teste anterior e possui, além dos seus próprios *test fixtures*, *test fixtures* do teste anterior.

Esta proposta pretende contribuir em contextos inspirados em histórias de teste. Histórias de teste, também conhecidas como teste de aceitação (Kamalrudin *et al.*, 2013), são testes de usuário utilizados com o objetivo de determinar se o sistema satisfaz ou não os critérios de aceitação definidos pelo cliente (Borg e Kropp, 2011). Em geral, testes de aceitação são especificados evolutivamente através de cenários de uso do sistema.

A proposta desse trabalho consiste em definir modelos que possam ser utilizados para a implementação de uma estratégia de *fixture setup* que permita que classes de teste possam utilizar *test fixtures* de outras classes de teste. Busca-se permitir criar encadeamentos lógicos de classes de teste, de modo que cada classe de teste possa representar, por exemplo, um passo de um cenário de uso do sistema. A ideia da proposta vem da simples observação de que a execução dos *test fixtures* de classe de teste pode levar o SUT exatamente para um estado necessário por uma outra classe de teste. É importante esclarecer que a definição explícita de dependência entre classes de teste proposta por esse trabalho não fere o princípio de independência dos testes, descrito no *Test Automation Manifesto* (Meszaros *et al.*, 2003). Abordaremos essa questão com mais detalhes adiante.

4.1. REUSO DE CÓDIGO DE TEST FIXTURES

Nesta seção serão apresentados modelos que servem de suporte para promover o reuso de código de *test fixtures*. Os modelos apresentados nesta seção servem de suporte também para promover o reuso de execução de *test fixtures*. Entretanto, o reuso de execução será abordado apenas na Seção 4.2.

4.1.1. Modelo de Dependência entre Classes de Teste

O modelo de dependência entre classes de teste propõe que uma classe de teste poderá depender de uma ou mais classes de teste. A relação dependência/dependente entre duas classes implica que a **classe consumidora** irá utilizar os *test fixtures* da **classe provedora**. A relação de dependência deverá ser definida na classe consumidora.

A Figura 6 mostra uma implementação do modelo de dependência utilizando a linguagem Java e o *framework* de teste Story que será explicado no Capítulo 5. A anotação `@FixtureSetup` é colocada na declaração da classe `ClasseY` para indicar uma dependência

com a classe `ClasseX`. A definição da dependência indica que a execução de um teste da classe `ClasseY` deverá utilizar, além dos *test fixtures* da própria classe `ClasseY`, os *test fixtures* da classe `ClasseX`. Os *test fixtures* são configurados na classe provedora e, posteriormente, são disponibilizados para a classe consumidora. A classe consumidora, por sua vez, poderá utilizar e até mesmo modificar os *test fixtures* disponibilizados. Com isso, torna-se possível promover o reuso de *test fixtures* entre classes de teste.

```
@FixtureSetup(ClasseX.class)
public class ClasseY {
  //...
}
```

Figura 6. Anotação `@FixtureSetup`.

4.1.1.1. Princípio da Independência

O princípio da independência, descrito no *Test Automation Manifesto* (Meszaros *et al.*, 2003), diz que cada teste deve ser independente. Deve ser possível executar cada teste individualmente ou através de uma suíte composta por um conjunto arbitrário de outros testes. Os testes de uma suíte devem poder ser executados em qualquer ordem. Esse princípio é justificado pelo fato de que a execução de um teste não deve influenciar na execução de outro. Através da justificativa, pode-se facilmente compreender porque o princípio foi adotado: um teste não deveria falhar apenas porque um teste anterior deixou o SUT em um estado inconsistente. Por isso, uma forma utilizada para evitar que a execução de um teste interfira na correta execução de outro teste consiste, antes do início de cada teste, em reiniciar o estado interno do SUT e reconfigurar os *test fixtures* necessários.

É necessário destacar que o princípio da independência de testes não é violado pelo modelo de dependência proposto neste trabalho. O modelo de dependência possibilita a definição de dependência entre classes de teste, e não entre testes. Os testes da classe consumidora dependem apenas dos *test fixtures*, e nunca dos testes, da classe provedora. Os *test fixtures* serão configurados novamente para cada um dos testes, ou seja, sempre que um teste for executado os *test fixtures* serão configurados novamente. Além disso, os testes poderão ser executados em qualquer ordem.

A relação que se espera entre classe consumidora e classe provedora pode ser melhor explicada através de um conceito da biologia. O comensalismo é uma relação entre dois organismos, onde um organismo se beneficia do outro sem prejudicá-lo (Beneden, 2009). É justamente um comportamento comensal que é esperado da relação de dependência entre duas classes de teste. Mais do que isso, é desejado que a codificação de uma classe de teste não precise levar em consideração a existência de eventuais classes consumidoras – apenas a classe consumidora deve conhecer a classe provedora. Objetiva-se, com isso, evitar, sempre que possível, o acoplamento entre classes de teste.

O princípio da independência não é violado nem mesmo quando o *test fixture* proveniente da classe provedora possui um erro. Nesse caso, o teste poderá falhar devido ao erro, porém é importante notar que esse problema ocorreria mesmo que fosse utilizada qualquer outra estratégia de *fixture setup* que promova o reuso de *test fixtures*.

4.1.1.2. Múltiplas Dependências

A Figura 7 apresenta um exemplo onde uma classe consumidora possui duas classes provedoras. Os *test fixtures* serão configurados na mesma ordem em que as classes provedoras aparecerem na anotação `@FixtureSetup`. A ordem em que os *test fixtures* são configurados é relevante, pois ordens diferentes podem levar o SUT a estados diferentes.

```
@FixtureSetup({
  ClasseX.class,
  ClasseY.class
})
public class ClasseZ {
  //...
}
```

Figura 7. Anotação `@FixtureSetup` com múltiplas dependências.

Múltiplas dependências trazem uma vantagem importante: *test fixtures* de classes diferentes podem ser combinados para constituir um novo *test fixture*. O modelo de dependência permite que uma classe consumidora utilize *test fixtures* distintos e de classes provedoras diferentes sem que seja necessário modificar as classes envolvidas. Essa estratégia pode ser adotada quando se deseja conservar a estrutura das

classes e, ao mesmo tempo, promover o reuso de *test fixtures* de duas ou mais classes diferentes.

Deve-se observar que as outras estratégias de *fixture setup* não conseguem, individualmente, contemplar o reuso de *test fixtures* e a conservação da estrutura das classes. O *inline setup* conserva a estrutura das classes, mas causa duplicação de código. O *implicit setup* promove o reuso de *test fixtures*, porém as classes precisam ser agrupadas em uma só. O *delegate setup* é o que, dentre as três estratégias, apresenta o melhor equilíbrio entre reuso de *test fixtures* e conservação da estrutura das classes. Para isso, os *test fixtures* devem ser movidos para métodos auxiliares. O conteúdo movido para os métodos auxiliares deixará de ser duplicado, porém, ainda que bem menores, existirão duplicações das chamadas para os métodos auxiliares. A conservação da estrutura das classes é parcialmente mantida. As classes de teste continuam sendo as mesmas, porém os *test fixtures* precisarão ser movidos do local onde foram originalmente definidos.

4.1.1.3. Dependências Transitivas

A relação de dependência entre as classes de teste é transitiva. Assim, é possível definir sequências de classes de teste encadeadas através da relação de dependência. A execução de um teste de uma classe consumidora deverá utilizar os *test fixtures* de todo o encadeamento de classes provedoras.

Sequências de classes de teste podem ser especialmente úteis para a implementação de testes de aceitação especificados evolutivamente. O modelo de dependência facilita a conversão de testes de aceitação evolutivos para código de teste. Em uma especificação evolutiva, testes de aceitação modelam cenários de uso do sistema, onde cada teste tem como ponto de partida um cenário anterior (Erdogmus, Morisio e Torchiano, 2005). De forma análoga, classes de teste materializariam testes de aceitação, onde cada classe teria como dependência a classe anterior.

Um problema das dependências transitivas está na possível ocorrência de ciclos. Uma classe de teste pode depender transitivamente de outra classe de teste que, por sua vez, depende transitivamente da primeira. A existência de uma dependência cíclica caracteriza um erro de compreensão do modelo de dependência, pois corresponde a um *loop* infinito durante a execução dos *test fixtures*.

4.1.2. Modelo de Manipulação de Test Fixture

O objetivo do modelo de manipulação é permitir que os *test fixtures* possam ser manuseáveis por testes de classes diferentes. Assim, testes de uma classe consumidora podem acessar *test fixtures* de classes provedoras. No modelo de manipulação isso deve ocorrer da seguinte maneira: *test fixtures* de classes provedoras devem ser atribuídos a um atributo da classe, enquanto a classe consumidora, por sua vez, deverá declarar, e apenas declarar, um atributo de mesmo nome. Assim, caberá ao *framework* de teste injetar os *test fixtures* das classes provedoras nos respectivos atributos das classes consumidoras.

A Figura 8 mostra uma implementação que faz uso do modelo de manipulação. Na classe `ClasseY` é declarado o atributo `fixtureX` que corresponde ao *test fixture* desejado. Na classe `ClasseX` o *test fixture* é configurado através do *setup method* e declarado como atributo da classe. Ao executar o teste, o *framework* deverá identificar a dependência e deverá injetar o *test fixture* `fixtureX` da classe provedora `ClasseX` na classe consumidora `ClasseY`.

```
@FixtureSetup(ClasseX.class)
public class ClasseY {

  @Fixture private Integer fixtureX;

  private Integer fixtureY;

  @Before
  public void configurar() {
    fixtureY = 2;
  }

  @Test
  public void testar() {
    assertEquals(1, fixtureX);
    assertEquals(2, fixtureY);
  }

}

public class ClasseX {

  private Integer fixtureX;

  @Before
  public void configurar() {
    fixtureX = 1;
  }

  // ...

}
```

Figura 8. Anotação `@Fixture`.

Apenas nomear o atributo da classe consumidora com o mesmo nome do *test fixture* da classe provedora já é suficiente para que o *framework* possa identificar quais *test fixtures* devem ser injetados. Entretanto, na implementação realizada neste trabalho optou-se por também anotar os atributos das classes consumidoras que representam *test fixtures* provenientes de classes provedoras. Na Figura 8 a anotação `@Fixture` indica ao *framework* de teste que o atributo anotado representa um *test fixture* proveniente de uma classe provedora. Essa abordagem apresenta duas principais motivações: (1) facilitar a identificação de atributos que representam *test fixtures* provenientes de classes provedoras; e (2) permitir que o *framework* de teste identifique eventuais inconsistências de nome do *test fixture*.

Conflitos de *test fixtures* com mesmo nome podem ocorrer quando uma classe consumidora possui múltiplas classes provedoras. Um conflito ocorre quando duas ou mais classes provedoras utilizam o mesmo nome para *test fixtures* diferentes. Nesse caso, o modelo de manipulação prevê que o *test fixture* injetado será aquele que for proveniente da classe provedora que for definida primeiro na declaração de dependência. Recomenda-se que o desenvolvedor evite esse tipo de cenário, pois pode dificultar a compreensão dos testes.

4.1.3. Utilização dos Modelos de Dependência e de Manipulação

A Figura 9 e a Figura 10 apresentam um exemplo onde, através dos modelos de dependência e manipulação propostos neste trabalho, é promovido o reuso de *test fixture* entre diferentes classes de teste. Na classe `TransientUserTest`, o teste `createUser` utiliza o *test fixture* `john` para verificar o comportamento de objetos da classe `User`. Já o teste da classe `PersistentUserTest` utiliza o *test fixture* `john` para verificar o comportamento de objetos da classe `User` após serem persistidos no banco de dados.

```
public class TransientUserTest {

  private User john;

  @Before
  public void setUp() {
    john = new User();
    john.setName("John");
    john.setCareer("Teacher");
  }

  @Test
  public void createUser() {
    assertNull(john.id());
    assertEquals("John", john.name());
    assertEquals("Teacher", john.career());
  }

}
```

Figura 9. Classe `TransientUserTest`.

```
@FixtureSetup(TransientUserTest.class)
public class PersistentUserTest {

  @Fixture private User john;

  private Integer id;

  @Before
  public void setUp() {
    UserDao userDao = new UserDao();
    id = userDao.insert(john);
  }

  @Test
  public void insertUser() {
    assertNotNull(id);
    assertEquals(id, john.id());
  }

}
```

Figura 10. Classe `PersistentUserTest`.

Pode-se perceber que existe uma relação temporal entra as duas classes apresentadas. A classe `PersistentUserTest` representa um cenário que é naturalmente posterior ao cenário representado pela classe `TransientUserTest`. Dessa forma, pode-se utilizar a proposta deste trabalho para que o *test fixture* `john` possa ser reutilizado entre as duas classes de teste. Através da anotação `@FixtureSetup`, o método `setUp` da classe `TransientUserTest` é incorporado, de forma transparente, à classe `PersistentUserTest`. A utilização da anotação `@Fixture` serve para indicar quais *test fixtures* da classe `TransientUserTest` deverão ser injetados na classe `PersistentUserTest`.

4.1.4. Grafo de Dependência e Grafo de Execução

O modelo de dependência permite que cada classe de teste possua múltiplas classes provedoras. Além disso, a relação de dependência é transitiva. Assim, é possível modelar as classes de teste e as relações de dependência através de um grafo direcionado. O grafo direcionado $\mathcal{G} = (\boldsymbol{V}, \boldsymbol{E})$ pode ser definido da seguinte forma:

$$\boldsymbol{V} = \{c \mid c \text{ é uma classe de teste do sistema}\}$$
$$\boldsymbol{E} = \{(c, d) \mid c \text{ é a classe consumidora e } d \text{ é a classe provedora}\}$$

Chamaremos $\mathcal{G}$ de **Grafo de Dependência (GD)**. O GD facilita a validação das relações de dependência entre classes de teste. Por exemplo, através de uma busca no GD, pode-se verificar se uma dada classe possui ou não dependência cíclica. O conjunto $\boldsymbol{V}$ contém todas as classes de teste do sistema. O GD será representado através de um diagrama de classes da UML (*Unified Modeling Language*), conforme apresentado na Figura 11. O estereótipo `<<fixture dependency>>` foi utilizado para indicar as relações de dependência entre os *test fixtures* de uma classe consumidora e os *test fixtures* da classe provedora. No exemplo apresentado, o teste da classe `A`, correspondente ao método `testA`, depende dos *test fixtures* configurados nas classes `B` e `C` que serão configurados, respectivamente, pelos métodos `setupB` e `setupC`. Por sua vez, os testes das classes `B` e `C`, correspondentes aos métodos `testB` e `testC`, respectivamente, dependem dos *test fixtures* configurados na classe `D` através do método `setupD`.

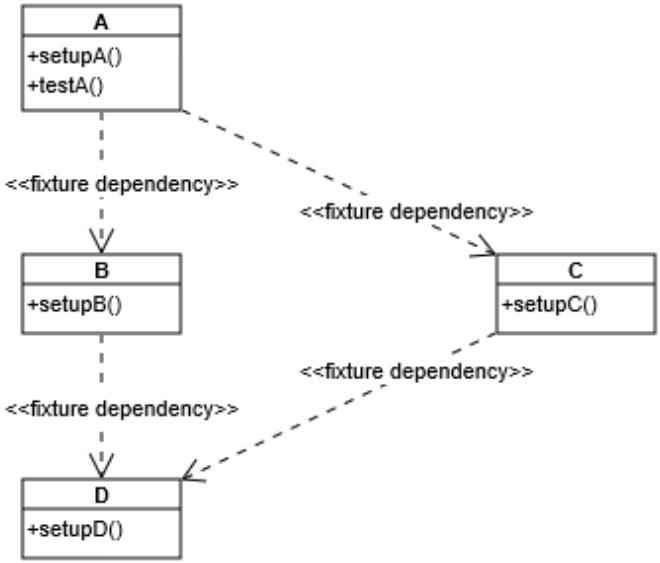


Figura 11. Grafo de Dependência.

Além de representar as dependências estáticas entre classes de teste, pode ser conveniente representar apenas as classes de teste que estarão envolvidas em uma dada execução de teste. Para isso, definiremos, a partir do GD, o subgrafo direcionado $\mathcal{H} = (\boldsymbol{W}, \boldsymbol{F})$:

$$\boldsymbol{W} = \{c \in \boldsymbol{V} \mid c \text{ é a classe em execução ou } c \text{ é uma classe provedora, direta ou transitiva, da classe em execução}\}$$

$$\boldsymbol{F} = \{(c, d) \mid c \text{ é a classe provedora e } d \text{ é a classe consumidora}\}$$

Chamaremos $\mathcal{H}$ de **Grafo de Execução (GE)**. Destaca-se que no GD a direção da aresta parte das classes consumidoras e aponta para as classes provedoras, enquanto no GE a direção da aresta é invertida, partindo das classes provedoras e apontando para as classes consumidoras, ou seja, enquanto no GD as relações entre os vértices representam a hierarquia de dependência, no GE estas mesmas relações representam a hierarquia de execução O GE também será representado através de um diagrama de classes da UML, conforme apresentado na Figura 12. O estereótipo `<<fixture execution dependency>>` é utilizado para indicar as relações de dependência entre a execução de *test fixtures* de uma classe provedora e a execução de *test fixtures* da classe consumidora. O estereótipo `<<running>>` é utilizado para indicar a classe de teste que se deseja executar. O GE apresentado na Figura 12 foi gerado considerando a execução do teste da classe `A` extraída a partir do GD apresentado na Figura 11. Assim, a classe `A` representa a classe que contém o teste que a será executado. As classes `B`, `C` e `D` representam as classes provedoras, diretas ou transitivas, da classe `A`.

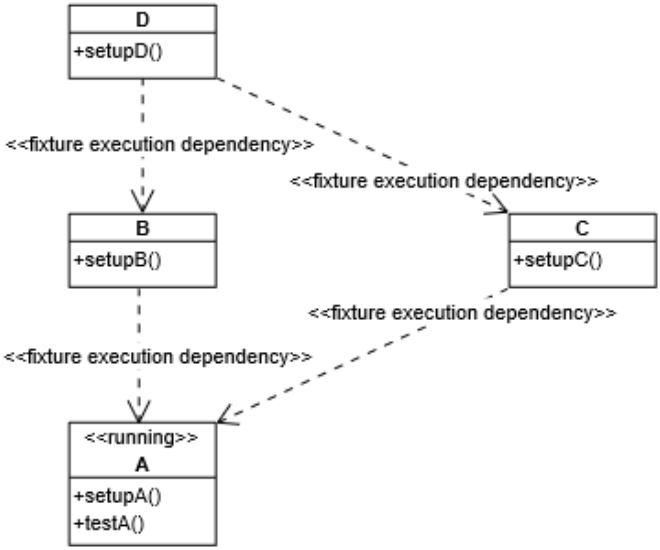


Figura 12. Grafo de Execução.

Percorrer o GE através de uma busca em largura passando por todos os vértices a partir da classe `D` possibilita determinar a **Sequência de Execução de *Test Fixtures* (SETF)**. A SETF determina a ordem em que os *test fixtures* devem ser executados para o teste. Entretanto, é necessário adotar uma estratégia para os casos em que duas ou mais classes consumidoras possuem uma mesma classe provedora em comum. Conforme mostra o diagrama de atividades da UML apresentado na Figura 13, a estratégia adotada pode ou não executar novamente os *test fixtures* da classe provedora. Sendo assim, existem duas SETF válidas e viáveis para o GE da Figura 12:

```
S1 = (setupD, setupB, setupD, setupC, setupA)
S2 = (setupD, setupB, setupC, setupA)
```

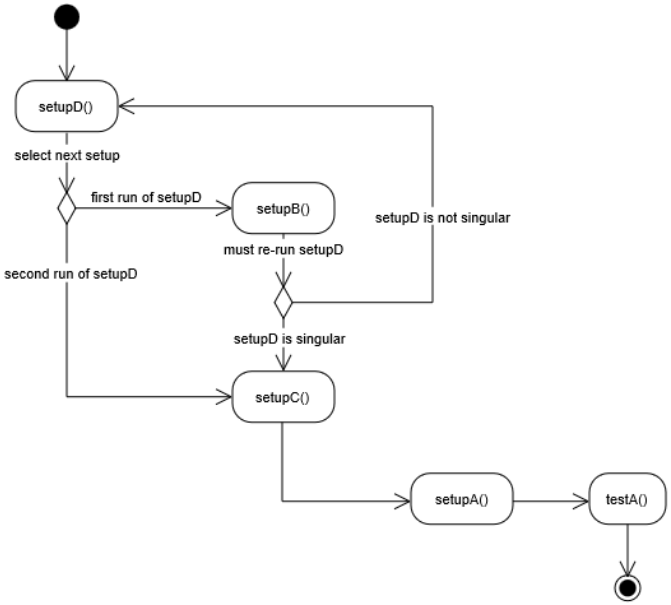


Figura 13. Definição das sequências de execução de *test fixtures*.

Em `S1`, a estratégia adotada inclui o vértice que representa a classe provedora antes de cada inclusão de vértice que representa uma classe consumidora. Na sequência `S1`, o valor do nó de decisão `must re-run setupD()` do diagrama de atividades da Figura 13 tem valor verdadeiro. Isso faz com que os *test fixtures* da classe `D` sejam executados novamente, dessa vez antes da execução dos *test fixtures* da classe `C`. Por outro lado, em `S2`, a estratégia adotada inclui o vértice que representa a classe provedora apenas uma vez, logo antes da inclusão do vértice que corresponde à primeira classe consumidora. Nesse caso, o valor do nó de decisão `must re-run setupD` é falso e, portanto, os *test fixtures* da classe D não serão executados novamente.

Existe, portanto, um impasse sobre como o *framework* de teste deve agir durante a execução. Esse tipo de impasse não pode ser resolvido pelo *framework*, pois existem casos onde é desejável que um *test fixture* seja executado repetidas vezes e existem casos onde não. Por isso, é necessário fazer uma distinção quanto à singularidade dos *test fixtures* de classes de teste. É necessário que seja especificado na classe provedora se os *test fixtures* contidos nela devem ser executados antes de cada execução dos *test fixtures* das classes consumidoras ou se devem ser executados apenas uma vez, antes da execução dos *test*

fixtures da primeira classe consumidora. Essa definição deverá ser feita pelo desenvolvedor do teste e é abordada com mais detalhes na Seção 4.1.5.

4.1.5. Modelo de Singularidade de Test Fixture

Um *test fixture* é dito singular quando deve ser executado uma e apenas uma vez durante um mesmo teste. Repetir a execução de um *test fixture* singular pode causar uma falha indesejada no teste.

No exemplo apresentado, anteriormente, na Figura 12, o *framework* de teste não pode determinar qual SETF deve executada, pois não tem como inferir se os *test fixtures* da classe `D` são ou não singular. Assim, torna-se necessário fazer uma adição ao modelo de dependência: todo *test fixture* de uma classe de teste será considerado não singular, a menos que o contrário tenha sido explicitamente definido na classe de teste. A Figura 14 mostra uma implementação em que o *test fixture* da classe de teste é explicitamente definido como singular. Para isso, anota-se a classe com a anotação `@Singular`.

```
@Singular
public class ClasseX {
  // ...
}
```

Figura 14. Anotação `@Singular`.

Assim, para uma mesma execução de teste, o *framework* deverá garantir que os *test fixtures* singulares sejam executados apenas uma vez. Os *test fixtures* não singulares deverão ser executados sempre que forem necessários por uma classe consumidora diferente.

4.2. REUSO DE EXECUÇÃO DE TEST FIXTURES

Até o presente momento este trabalho abordou o reuso de *test fixtures* a partir de uma perspectiva voltada para promover o reuso de código de teste. Isso contribuí para que sejam desenvolvidos testes mais robustos e, portanto, com menor custo de manutenção a longo prazo. Nesta seção, a proposta deste trabalho será complementada com o objetivo de promover o reuso de execução de *test fixtures*. Através do reuso de execução pode-se reduzir o tempo de execução dos testes.

4.2.1. Modelo de Segurança de Teste

Dizemos que um teste é **seguro** quando a execução de um teste não altera o estado dos *test fixtures*. Um teste é considerado **inseguro** quando sua execução "suja" *test fixtures* que não foram configurados dentro do próprio método de teste (isto é, não foram criados através da estratégia *inline setup*). O efeito colateral de testes inseguros é que os *test fixtures* não podem ser utilizados coletivamente por esses testes. Deixar um *test fixture* coletivo "sujo" pode causar uma falha indesejada em um próximo teste que utilizar o mesmo *test fixture*, uma vez que o *test fixture* será deixado em um estado diferente daquele esperado pelo próximo teste.

A Figura 15 mostra uma classe com dois testes, um seguro e outro inseguro. No exemplo os dois testes utilizam o *test fixture* chamado `fixture`, que representa uma lista de *strings*. O primeiro teste é dito seguro, pois não modifica o *test fixture* da classe. Já o segundo teste é digo inseguro, uma vez que altera o valor inicial do *test fixture*. Nesse caso, o teste inseguro altera o *test fixture* ao adicionar a *string* "b" na lista. Isso significa que o *test fixture* não pode ser utilizado de forma coletiva pelo segundo teste, pois se isso acontecer, os próximos testes que utilizarem o *test fixture* poderão falhar.

```
public class ClasseX {

  private List<String> fixture;

  @Before
  public void configurar() {
    fixture = new ArrayList<>();
    fixture.add("a");
  }

  @Test
  public void testar1() {
    assertEquals(1, fixture.size());
    assertEquals("a", fixture.get(0));
  }

  @Test
  public void testar2() {
    fixture.add("b");
    assertEquals(2, fixture.size());
    assertEquals("a", fixture.get(0));
    assertEquals("b", fixture.get(1));
  }

}
```

Figura 15. Teste seguro e teste inseguro.

Uma alternativa para possibilitar que testes inseguros utilizem *test fixtures* coletivos é garantir que a execução dos testes inseguros seja realizada por último. Assim, o *test fixture* "sujo" poderá ser sujo pelo teste sem problemas, uma vez que existe garantia que mais nenhum teste utilizará aquele *test fixture*. Tradicionalmente, *frameworks* de teste não fazem distinção entre testes seguros e inseguros. Esse é um dos motivos pelo qual as estratégias de *shared fixture construction* tipicamente não são embutidas de forma nativa nos *frameworks* de teste. Distinguir entre testes seguros e inseguros pode facilitar a inclusão desse tipo de estratégia permitindo que se utilize *test fixtures* coletivos.

Por exemplo, considerando que na Figura 15 se possa distinguir entre testes seguros e inseguros, através dessa distinção seria possível utilizar uma estratégia de *fixture setup* onde a execução dos testes seria ordenada de modo que primeiro fosse executado o teste seguro e por último o teste inseguro. Assim, o *test fixture* poderia ser utilizado de forma coletiva, mesmo que exista um teste inseguro. Utilizar *test fixtures* coletivos pode reduzir o tempo de execução dos testes uma vez que o *test fixture* precisará ser executado com menos frequência.

Um princípio básico da atividade de teste é que deve ser possível executar todos os testes em uma ordem arbitrária qualquer (Meszaros *et al.*, 2003). Isso significa que um teste não deve poder definir a sua própria ordem de execução. Entretanto, um *framework* de teste pode definir a ordenação da execução dos testes na forma em que for mais conveniente.

A Figura 16 apresenta uma proposta de implementação para distinguir entre testes seguros e inseguros. Através das anotações `@Safe` e `@Unsafe`, o desenvolvedor do teste pode informar ao *framework* se um teste anotado é seguro ou não. O *framework* de teste pode também considerar um valor padrão para quando a anotação não é utilizada. Por exemplo, pode-se considerar que um teste é sempre inseguro, a menos que exista uma declaração explícita dizendo que é seguro.

```
public class ClasseX {

  @Safe
  @Test
  public void testar1() {
    // ...
  }

  @Unsafe
  @Test
  public void testar2() {
    // ...
  }

}
```

Figura 16. Anotação `@Safe`.

4.2.2. Coletivização de Test Fixture

Conforme descrito na seção anterior, classificar testes como seguros ou inseguros possibilita coletivizar *test fixtures*. Se todos os testes que serão executados forem seguros, então os *test fixtures* que forem idênticos poderão ser coletivizados entre todos os testes. Até então, este trabalho considerou apenas a execução de um único teste. Considerando a execução de um único teste, apenas pode-se promover o reuso de código de *test fixtures*. Entretanto, quando são consideradas execuções de diferentes testes, passa a ser possível promover o reuso de execução de *test fixtures*. O potencial de reuso de execução irá variar conforme o número de *test fixtures* coletivos que podem ser utilizados. Quanto maior forem esse número, maior será o potencial de reuso.

O modelo de dependência traz algumas informações importantes que permite maximizar o número de *test fixtures* coletivos. A declaração de dependência entre duas classes indica que a classe consumidora irá utilizar *test fixtures* da classe provedora. Isso significa que dependendo dos testes executados, os *test fixtures* da classe provedora poderão ser coletivizados. Por exemplo, considerando um longo encadeamento de classes de teste dependentes. Se cada classe possuir um único teste seguro, então os *test fixtures* executados para o teste da primeira classe poderão ser coletivizados para os testes seguintes do encadeamento.

A seguir, será mostrado como as informações do modelo de dependência e do modelo de segurança podem ser utilizadas por um *framework* de testes para coletivizar *test fixtures* e, assim, promover o reuso de execução. Para isso, serão apresentados alguns cenários. Cada

cenário contém uma suíte de teste composta por classes de teste. Além disso, em cada cenário dois conjuntos serão apresentados: o conjunto de **Sequências de Execução (SE)** e o conjunto de **Sequências de Execução com Coletivização (SEC)**. A diferença entre os dois conjuntos está no fato de que no conjunto SEC os *test fixtures* são coletivizados durante a execução da suíte de teste.

Os cenários são baseados no GD (Grafo de Dependência, ver Seção 4.1.4) apresentado na Figura 17. Cada vértice no GD corresponde a uma classe de teste. Cada classe de teste possui métodos de configuração e métodos de teste. Métodos de teste inseguros são marcados através do estereótipo `<<unsafe>>`, enquanto métodos de teste seguros são marcados através do estereótipo `<<safe>>`.

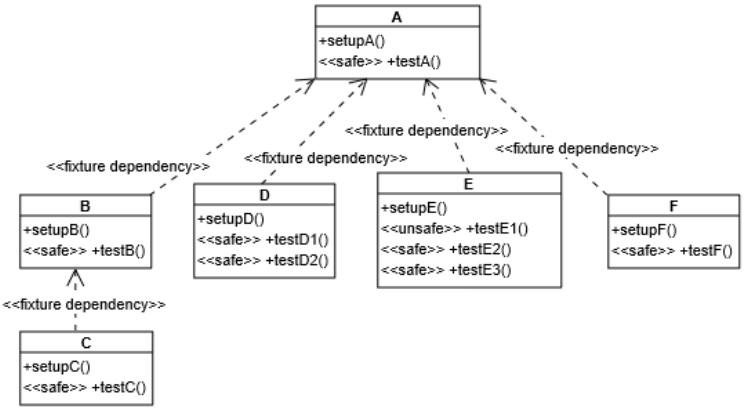


Figura 17. Grafo de Dependência.

O somatório das diferenças entre a quantidade de execuções dos conjuntos SE e SEC será chamado doravante de **Índice de Reuso de Execução (IRE)**. Este índice será utilizado para avaliar a diferença entre o reuso de execução de *test fixtures* com e sem a proposta.

4.2.2.1. Execução de todas classes de um ramo do GD

Esse cenário mostra a execução de um ramo inteiro do GD onde todos testes são seguros. Esse é o melhor cenário possível em que a coletivização de *test fixtures* pode ser aplicada. Considerando a imagem apresentada na Figura 17, deseja-se executar os testes das classes `A`, `B` e `C`. As classes de teste mencionadas pertencem a um mesmo ramo do GD. Abaixo são descritos os conjuntos SE e SEC. Nas nomenclaturas

`SE[N]` e `SEC[N]` utilizadas, `N` representa a enésima sequência de execução do conjunto. No caso do conjunto SE, cada sequência corresponderá à sequência de execução para um único caso de teste. Este é o cenário tradicional, onde não ocorre o reuso de execução de *test fixtures*. Já no conjunto SEC, uma sequência poderá conter a execução de mais de um teste, caso onde o reuso de execução de *test fixtures* acontece.

```
Suíte = (A, B, C)

SE[1] = (setupA, testA)
SE[2] = (setupA, setupB, testB)
SE[3] = (setupA, setupB, setupC, testC)

SEC[1] = (setupA, testA, setupB, testB, setupC, testC)
```

Nesse cenário, a suíte corresponde a todas as classes de teste de um determinado ramo do GD. Além disso, todos os testes a serem executados são seguros. Essas características tornam possível maximizar o reuso de execução. Ao invés de executar três sequências separadas (SE), o *framework* de teste pode ordenar os testes e então juntar as três sequências em uma só (SEC).

No conjunto SE, o *setup method* contendo os *test fixtures* da classe `A` é executado três vezes e o *setup method* da classe `B` é executado duas vezes, enquanto no SEC, como os testes são ordenados de acordo com a dependência das classes de teste, os *setups methods* precisam ser executados apenas uma vez cada.

O valor de IRE para este caso foi de 3, uma vez que esse valor foi a diferença entre o número de execuções de *setup methods* dos conjuntos SE e SEC. Em cenários análogos a este, o valor do IRE será relativo ao tamanho do ramo do GD executado. Por exemplo, se existisse uma outra classe no ramo, então o IRE pularia de 3 para 6. Para este tipo de cenário o IRE pode ser calculado pela soma de uma progressão aritmética de razão 1 e termo inicial igual a 0. O cálculo pode ser feito através da fórmula apresentada abaixo onde `h` é o tamanho do ramo.

```
IRE = h * (h -1) / 2
```

É importante destacar que esta fórmula pode ser utilizada apenas quando as seguintes condições foram satisfeitas: (1) todos os testes de

todas as classes de um determinado ramo devem ser executados; (2) toda classe do ramo possui apenas um *setup method*; e (3) todo teste do ramo é seguro.

4.2.2.2. Execução de algumas classes de um ramo do GD

Este cenário é similar ao anterior com a diferença que nem todas as classes de um ramo são executadas. Apesar disso, ainda é possível promover o reuso de execução de *test fixtures*.

```
Suíte = (A, C)

SE[1] = (setupA, testA)
SE[2] = (setupA, setupB, setupC, testC)

SEC[1] = (setupA, testA, setupB, setupC, testC)
```

Neste cenário, apenas a coletivização do *test fixture* da classe `A` pode ser aproveitada para promover o reuso de execução. Nesse caso, o IRE é igual a 1, pois a quantidade de execuções de *setup methods* do SE é 5 e a quantidade de execuções de *setup methods* do SEC é 5. Deve ser observado que, diferente do cenário anterior, neste cenário a classe B não foi incluída na suíte e por isso o reuso de execução foi menor. Ainda assim, deve-se destacar que o reuso de execução foi ótimo, isto é, o maior possível para a suíte em questão.

4.2.2.3. Execução de classes com mais de um teste seguro

Este cenário mostra o IRE será tão maior quanto for a quantidade de testes seguros de uma classe.

```
Suíte = (D)

SE[1] = (setupA, setupD, testD1)
SE[2] = (setupA, setupD, testD2)

SEC[1] = (setupA, setupD, testD1, testD2)
```

A classe CD utilizada neste cenário possui dois testes seguros. Como ambos os testes são seguros, garante-se que os *test fixtures* podem ser coletivizados entre os dois testes da classe. Com isso, torna-se possível que os *setups methods* sejam executados apenas uma vez para

os dois testes. O IRE alcançado foi 2, uma vez que no SE a quantidade de execuções de *setup methods* foi 4 para o conjunto SE e 2 para o conjunto SEC. Destaca-se que quanto maior for o número de testes seguros e *setup methods* em uma classe, maior será o IRE. Através da seguinte fórmula pode-se calcular o IRE para cenários análogos a este onde as seguintes condições são satisfeitas: (1) todos os testes são seguros; e (2) todos os testes utilizam a mesma quantidade de *setup methods*. Na fórmula a seguir `f` é a quantidade de *setup methods* e `t` é a quantidade de testes:

```
IRE = f * (t - 1)
```

4.2.2.4. Execução de classes com testes inseguros

Neste cenário será analisado o reuso de execução de *test fixtures* quando uma classe contém, ambos, testes seguros e inseguros. Será mostrado que mesmo com testes inseguros é possível coletivizar os *test fixtures* entre alguns testes da suíte promovendo, dessa forma, o reuso de execução.

```
Suíte = (CE)

SE[1] = (setupA, setupE, testE1)
SE[2] = (setupA, setupE, testE2)
SE[3] = (setupA, setupE, testE3)

SEC[1] = (setupA, setupE, testE1)
SEC[2] = (setupA, setupE, testE2, testE3)
```

Este cenário é similar ao anterior com a diferença de que a classe CE tem também um teste inseguro. O cálculo do IRE pode utilizar a mesma fórmula do cenário anterior, desde que seja subtraído de `t` a quantidade de testes inseguros. Assim, o IRE calculado para este cenário é de 2, pois o número de execuções de *setup methods* foi 6 para o conjunto SE e 4 para o conjunto SEC.

Conforme discutido anteriormente, não é possível coletivizar *test fixtures* entre testes inseguros. Isso se deve ao fato de que testes inseguros podem alterar o estado dos *test fixtures* deixando-os em um estado inesperado para um eventual próximo teste. Entretanto, considerando que exista apenas um teste inseguro, então pode-se ordenar a execução de tal forma que o último teste executado seja o teste inseguro. Desse modo, o teste inseguro poderá utilizar e “sujar” o *test*

fixture coletivo sem que isso cause uma falha indesejada em outro teste, uma vez que não haverá mais testes a serem executados. Nesse caso, um IRE de 3 poderia ser alcançado através da utilização do seguinte SEC:

```
SEC[1] = (setupA, setupE, testE2, testE3, testE1)
```

4.2.2.5. Execução de classes de diferentes ramos do GD

Neste cenário é mostrado que não podem ser utilizados *test fixtures* coletivos entre testes de classes de diferentes ramos do GD.

```
Suíte = (B, F)

SE[1] = (setupA, setupB, testB)
SE[2] = (setupA, setupF, testF)

SEC[1] = (setupA, setupB, testB)
SEC[2] = (setupA, setupF, testF)
```

A coletivização de *test fixtures* pode ser aplicada para os testes de classes que estão em um mesmo ramo do GD. Entretanto, isso não é possível para os testes de classes que estão em diferentes ramos do GD, nem mesmo se todos os testes forem seguros. Assim, como as classes `B` e `F` estão em diferentes ramos, o SEC é igual ao SE e o IRE tem valor igual a 0.

Deve-se destacar que as duas classes de teste da suíte possuem a classe `A` como classe provedora. Apesar de neste caso ser promovido o reuso de código de *test fixtures*, isso não implica que o reuso de execução de *test fixtures* também seja possível. Um teste seguro é aquele que não suja os *test fixtures* da própria classe ou de classes provedoras. Mesmo para um teste declarado como seguro, não existe garantia de que o teste não irá "sujar" algum *test fixture* de classes de diferentes ramos do GD. Quando um teste é declarado como seguro garante-se apenas que o teste não irá "sujar" os *test fixtures* conhecidos. Não se pode garantir que um teste seguro não "suje" *test fixtures* de classes de diferentes ramos do GD, pois tais classes podem nem existir no momento em que o teste é criado.

Neste cenário, o impedimento para a coletivização dos *test fixtures* não ocorre porque os testes das classes `B` ou `F` podem "sujar" os *test fixtures* da classe `A`, mas sim porque o teste da classe `B` pode "sujar" o *test fixture* da classe `F`, ou vice-versa. Dessa forma, para garantir que o

princípio da independência (ver Seção 4.1.1.1) seja respeitado, a coletivização dos *test fixtures* não deve ser realizada.

4.3. DISCUSSÃO

Neste capítulo foi apresentada a proposta deste trabalho. Mostrou-se, através da proposição de modelos, que é possível promover o reuso de *test fixtures* entre classes de teste sem criar dependência entre os testes. Através do modelo de dependência, um teste pode utilizar de maneira confiável *test fixtures* definidos em outras classes de teste. Além do reuso de código de *test fixture*, os modelos propostos também permitem o reuso de execução. Para promover o reuso de execução, no entanto, é necessário que seja feita uma distinção entre testes seguros e inseguros. Essa distinção deve ser explicitada, através de anotações, no próprio método de teste.

5. FRAMEWORK STORY

Este capítulo apresenta o Story, um *framework* desenvolvido para dar suportar aos modelos apresentados nesta proposta. O Story foi construído com base na arquitetura do *framework* de teste JUnit. O *framework* criado incorpora os modelos propostos e mantém a compatibilidade com testes desenvolvidos para o JUnit. Assim, as mesmas anotações utilizadas em testes do JUnit funcionam para os testes do Story. Além disso, foram adicionas novas anotações que permitem ao *framework* suportar os modelos propostos neste trabalho. A anotações são as mesmas utilizadas nos exemplos apresentados anteriormente: `@FixtureSetup`, `@Fixture`, `@Singular` e `@Safe`.

5.1. DESENVOLVIMENTO DO FRAMEWORK

O Story foi desenvolvido através da abordagem *Test-Driven Development* (TDD) (Beck, 2002) e seu código foi disponibilizado de forma aberta através da plataforma GitHub[4]. O desenvolvimento resultou na implementação de 38 classes de produção e de 223 testes distribuídos através de 28 classes de teste. Apesar

Para o desenvolvimento foram utilizados: o ambiente de desenvolvimento integrado (*Integrated Development Environment* – IDE) Eclipse, a linguagem de programação Java e o sistema de controle de versão Git. A única dependência do Story é o *framework* de teste JUnit – não foi utilizado nenhum outro *framework* ou biblioteca, além do JUnit, para o desenvolvimento do Story.

O Story possui integração com o Eclipse. Neste ambiente de desenvolvimento testes com o Story podem ser executados através dos mesmos mecanismos (atalhos, configurações de execução de testes e outros) utilizados para executar testes com o JUnit. Não foi realizada integração com outros ambientes de desenvolvimento além do Eclipse.

5.2. PRÉ-EXECUÇÃO DOS TESTES

O Story utiliza o Grafo de Dependência (ver Seção 4.1.4) para representar e validar as relações de dependência. Assim, são realizadas verificações sobre as classes de testes com a finalidade de procurar eventuais inconsistências na utilização dos modelos propostos neste

[4] https://github.com/lucasPereira/estoria

trabalho, como, por exemplo, a existência de um ciclo de dependência entre classes de teste.

Antes da execução dos testes, o Story também verifica os *test fixtures* que são importados de outras classes de teste. Atributos da classe de teste com a anotação `@Fixture` devem existir nas classes provedoras. Com isso, a utilização da anotação `@Fixture` permite que o Story verifique dinamicamente, antes de executar os testes, se foi declarado em alguma classe de teste um *test fixture* que na verdade não existe. Essa verificação é particularmente útil para detectar eventuais inconformidades nos nomes dos *test fixtures*.

5.3. CLASSES

O Story foi desenvolvido a partir do *framework* de teste JUnit. Apesar disso, o Story controla o próprio fluxo de execução. O JUnit foi utilizado, basicamente, como uma biblioteca de classes. Por esse motivo, a maior parte Story precisou ser implementada a partir do zero.

A Figura 18 apresenta um diagrama de classes onde são mostradas as classes do JUnit que foram utilizadas pelo Story. Também são mostradas no diagrama as principais classes do Story, em especial destacam-se aquelas que estendem classes do JUnit.

A classe `StoryRunner`, que estende a classe abstrata `Runner` do JUnit, sobrescreve o método `run`, herdado de `Runner`, para que possa gerenciar o fluxo de execução dos testes. Já a classe `StoryBuilder`, que estende a classe abstrata `RunnerBuilder` do JUnit, sobrescreve o método `runnerForClass`, herdado de `RunnerBuilder`, para que instâncias da classe `StoryRunner` possam ser construídas. Além disso, a classe `StoryRunner` implementa a interface `Filterable` do JUnit. Através desta interface é possível aplicar filtros para impedir que alguns métodos sejam executados. Isso é particularmente útil quando se deseja executar um teste de uma classe sem que os outros testes da classe também sejam executados. Em geral, este tipo de filtro é aplicado pelo ambiente de desenvolvimento onde os testes estão sendo executados.

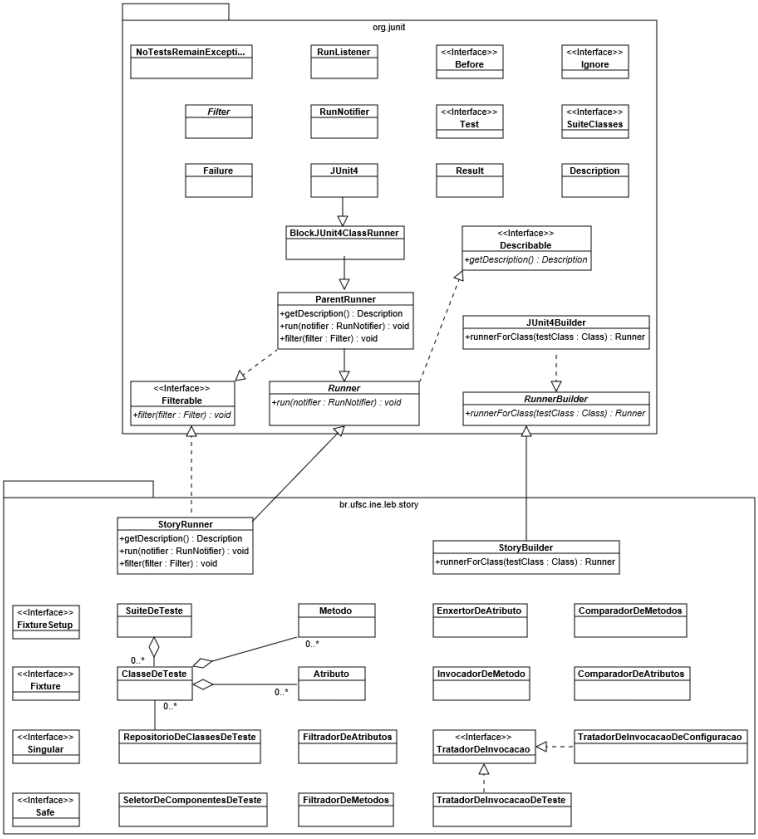


Figura 18. Diagrama de classes do JUnit e Story.

Além das classes que já foram mencionadas, destacam-se também as classes `Description`, `RunNotifier` e `RunListener`, do JUnit. A classe `Description` contém a descrição da suíte de teste que será executada. Nela são descritos os métodos e classes de testes. No JUnit uma `Description` é construída e fornecida pelo `Runner`, mas no caso do Story isto é realizado na classe `StoryRunner`. A classe `RunNotifier` é utilizada pela classe `StoryRunner` para notificar eventos durante a execução dos testes. Em especial, chama-se a atenção para os métodos `fireTestStarted`, usado para notificar que a execução de um dado teste foi iniciada, `fireTestFailure`, usado para notificar que um dado teste falhou, `fireTestIgnored`, usado para notificar que um dado teste não foi executado, e `fireTestFinished`, usado para

notificar que a execução de um dado teste foi finalizada. As notificações enviadas pela classe `StoryRunner` para a classe `RunNotifier` são, então, despachadas para instâncias da classe `RunListener`. A classe `RunListener` não foi utilizada pelas classes de produção do Story, ela foi utilizada apenas pelas classes de teste para que o Story pudesse ser testado.

A inicialização da execução dos testes é realizada, em geral, pela própria IDE. A IDE utiliza-se da classe `RunnerBuilder` para criar uma instância da classe `Runner`. No caso do Story, deve ser utilizada a classe `StoryBuilder` para criar uma instância da classe `StoryRunner`. A inicialização da execução dos testes com o Story ocorre da seguinte maneira: (1) a IDE faz uma chamada para o método `runnerForClass` da classe `StoryRunner` enviando como parâmetro a classe de teste que deverá ser executada; (2) a classe `StoryBuilder` constrói uma instância da classe `StoryRunner` enviando como parâmetro do construtor a mesma classe de teste recebida no passo 1; (3) uma instância da classe `Description` é construída pela instância da classe `StoryRunner` construída no passo 2; (4) a classe `StoryBuilder` retorna para a IDE a instância da classe `StoryRunner` construída no passo 2; (5) a IDE constrói uma instância da classe `RunListener`; (6) a IDE constrói uma instância da classe `RunNotifier` e registra nesta última a instância da classe `RunListener` construída no passo 5; (7) a IDE utiliza a instância da classe `StoryRunner` construída no passo 2 para chamar o método `run` enviando como parâmetro a instância da classe `RunNotifier` construída no passo 6; (8) a execução dos testes é, então, iniciada pela instância da classe `StoryRunner` construída no passo 2 que irá enviar as notificações sobre a execução através da instância da classe `RunNotifier` construída no passo 6 que, por sua vez, irá despachar as notificações para a instância da classe `RunListener` construída no passo 5; (9) a IDE utiliza a instância da classe `Description` construída no passo 3 e retornada pelo método `getDescription` da instância da classe `StoryRunner` construída no passo 2 para exibir a descrição da suíte de testes em execução; e (10) a IDE utiliza a instância da classe `RunListener` construída no passo 5 para monitorar as notificações enviadas no passo 8 e, assim, exibir os resultados dos testes.

5.4. FLUXO DE EXECUÇÃO

No JUnit, o fluxo de execução dos testes é definido através de implementações da classe abstrata `Runner`. Através do método `run`, as implementações devem definir o fluxo de execução dos testes. O JUnit oferece uma implementação padrão de `Runner`. Entretanto, para que seja possível implementar os modelos propostos foi preciso controlar o fluxo de execução dos testes, sobretudo o momento em que os *test fixtures* são executados. Assim, para a criação do *framework* Story foi criada uma nova implementação da classe `Runner`, a subclasse `StoryRunner`. Essa implementação é responsável pelas seguintes etapas: (1) executar os *test fixtures* de classes provedoras; (2) executar os *test fixtures* da classe em execução; (3) injetar os *test fixtures* nas classes consumidoras; (4) executar o teste; e (5) verificar e registrar o resultado da execução do teste.

O `Runner` padrão do JUnit realiza as etapas 2, 4 e 5. Entretanto, para possibilitar a inclusão das etapas 1 e 3 foi necessário criar um `Runner` completamente novo.

A Figura 19 mostra, em pseudocódigo, o algoritmo básico utilizado pelo Story. Por padrão o algoritmo apresentado promove o reuso de execução de *test fixtures*. Para o Story, todos os testes são potencialmente inseguros. Assim, todo teste é considerado inseguro, a menos que o teste seja anotado com a anotação `@Safe`. Desenvolvedores de teste devem utilizar esta anotação para indicar quais testes podem utilizar *test fixtures* coletivos. Caso não seja desejado utilizar o reuso de execução, então basta que nenhum teste seja anotado com a anotação `@Safe`.

```
runSuite : (suite)
  executed: {}
  for each leafTestClass in suite
    runBranch(leafTestClass, executed)
  for each unsafeTestMethod in suite
    runTestMethod(unsafeTestMethod)

runBranch : (testClass, executed)
  initializeAttributes(testClass)
  for each providerClass in testClass
    runBranch(providerClass, executed)
  injectFixtures(testClass)
  runSetupMethods(testClass)
  for each safeTestMethod in testClass
    if safeTestMethod is not in executed
      safeTestMethod()
      executed <- safeTestMethod

runTestMethod : (unsafeTestMethod)
  testClass: unsafeTestMethod.testClass
  initializeAttributes(testClass)
  runTestFixtures(testClass)
  unsafeTestMethod()

runTestFixtures: (testClass)
  for each providerClass in testClass
    runFixtureSetups(providerClass)
  injectFixtures(testClass)
  runSetupMethods(testClass)

runSetupMethods : (testClass)
  for each setupMethod in testClass
    setupMethod()

initializeAttributes : (testClass)
  for each attribute in testClass
    attribute <- null

injectFixtures : (testClass)
  injected: {}
  for each fixture in testClass
    for each providerClass in testClass
      for each attribute in providerClass
        if (fixture is compatible with attribute and fixture is not in injected)
          fixture <- attribute
          injected <- fixture
```

Figura 19. Algoritmo de execução do Story.

A execução do algoritmo deve ser iniciada pela função `runSuite` e seu comportamento é explicado como segue. Primeiramente, o algoritmo busca pelos nós folha no Grafo de Dependência (ver Seção 4.1.4). Cada nó folha corresponde a um ramo de dependência de classes de teste. Os testes seguros de cada ramo são executados ordenadamente, iniciando pelos testes da classe correspondente ao nó raiz e terminando pelos testes da classe correspondente o nó nodo folha. Para cada classe em um dado ramo, o algoritmo irá executar os *test fixtures* e os testes seguros da classe, e, então, irá remover os testes seguros executados da lista de testes pendentes para execução. Somente após executar os testes seguros de todos os ramos é que o algoritmo irá executar os testes inseguros onde não se pode utilizar *test fixtures* coletivos.

6. AVALIAÇÃO

Neste capítulo a aplicabilidade da proposta é avaliada através de um exemplo de uso e de experimentos. Segundo Rombach *et al.* (2006), em estudos empíricos é difícil comparar novas técnicas com outras já existentes. Não se pode esperar perfeição ou respostas decisivas. Entretanto, através da observação de variáveis independentes, pode-se obter informações valiosas. Por isso, além da apresentação de um exemplo de uso, foram realizados experimentos através dos quais buscou-se avaliar o reuso de código e o reuso de execução. Esses experimentos são importantes, pois possibilitam, além da avaliação da aplicabilidade da proposta, o estabelecimento de uma base de comparação entre a estratégia proposta e as estratégias convencionais.

6.1. EXEMPLO DE USO

Nesta seção será apresentado um exemplo de uso onde a proposta deste trabalho foi aplicada. Este exemplo tem como objetivo ilustrar o funcionamento dos modelos propostos neste trabalho apenas em relação ao reuso de código de *test fixtures*. O reuso de execução de *test fixture* não será, portanto, abordado ao longo deste exemplo. O exemplo apresentado foi adaptado a partir dos testes de um sistema de escalonamento de horários para eventos. No sistema podem ser criados eventos onde cada um pode ter a participação de diversos usuários e cada usuário deve definir os horários em que pode participar do evento. Apenas os testes para as entidades usuário e evento foram considerados nesse exemplo. A classe de teste apresentada na Figura 20 contém testes para a entidade `User` e a classe de teste apresentada na Figura 21 contém testes para a entidade `Event`. A classe de teste apresentada na Figura 22 contém os testes da entidade associativa `UserEvent` que representa a relação entre um evento e um usuário.

```
@FixtureSetup(NoDataTest.class)
public class UserTest {

  @Fixture private UserDao userDao;

  private Long id;
  private User john;

  @Before
  public void createAndInsertJohn() {
    john = new User();
    john.setName("John");
    john.setCareer("Teacher");
    id = userDao.insert(john);
    assertNotNull(id);
  }

  @Test
  public void get() {
    User user = userDao.getEntity(id);
    assertEquals(id,user.id());
    assertEquals(john.id(),user.id());
    assertEquals("John",user.name());
    assertEquals("Teacher",user.career());
  }

  @Test
  public void list() {
    List<User> list = userDao.list();
    assertEquals(1,list.size());
    User user = list.get(0);
    assertEquals(id,user.id());
    assertEquals(john.id(),user.id());
    assertEquals("John",user.name());
    assertEquals("Teacher",user.career());
  }

}
```

Figura 20. Classe `UserTest`.

```
@FixtureSetup(NoDataTest.class)
public class EventTest {

  @Fixture private EventDao eventDao;

  private Long id;
  private Event lecture;

  @Before
  public void configure() {
    lecture = new Event();
    lecture.setName("Lecture");
    id = service.insert(lecture);
    assertNotNull(id);
  }

  @Test
  public void get() {
    Evento event = eventDao.entity(id);
    assertEquals(id,event.id());
    assertEquals(lecture.id(),event.id());
    assertEquals("Lecture",event.name());
  }

  @Test
  public void list() {
    List<Event> list = eventDao.list();
    assertEquals(1,list.size());
    Evento event = list.get(0);
    assertEquals(id,event.id());
    assertEquals(lecture.id(),event.id());
    assertEquals("Lecture",event.name());
  }

}
```

Figura 21. Classe `EventTest`.

```
@FixtureSetup({
  UserTest.class,
  EventTest.class
})
public class UserEventTest {

  @Fixture private User john;
  @Fixture private Event lecture;
  @Fixture private UserEventDao dao;

  private Long id;
  private UserEvent johnLecture;

  @Before
  public void configure() {
    johnLecture = new UserEvent();
    johnLecture.setUser(john);
    johnLecture.setEvent(lecture);
    id = dao.insert(johnLecture);
    assertNotNull(id);
  }

  @Test
  public void get() {
    UserEvent entity = dao.entity(id);
    User user = entity.user();
    Event event = entity.event();
    assertEquals(id,entity.id());
    assertEquals(johnLecture,entity.id());
    assertEquals(john.id(),user().id());
    assertEquals(lecture.id(),
    event().id());
  }

  @Test
  public void list() {
    List<UserEvent> list = dao.list();
    assertEquals(1,list.size());
    UserEvent entity = list.get(0);
    User user = entity.user();
    Event event = entity.event();
    assertEquals(id,entity.id());
    assertEquals(johnLecture,entity.id());
    assertEquals(john.id(),user().id());
    assertEquals(lecture.id(),
    event().id());
  }

}
```

Figura 22. Classe `UserEventTest`.

O *test fixture* da classe `UserTest`, mostrada na Figura 20, é realizado através do *setup method*, método `createAndInsertJohn`. O *test fixture* para os dois testes da classe consiste em ter somente a entidade representada pelo objeto `john` cadastrada no banco de dados. Para a execução dos *test fixtures* dos testes da classe é necessário que se tenha como ponto de partida um cenário onde não exista nenhuma entidade já persistida no banco de dados. Esse cenário é desejado para promover um maior controle sobre os testes e evitar a ocorrência de testes frágeis. Para garantir o cenário desejado, existem duas abordagens possíveis: (1) limpar todo o banco de dados antes de cada teste; ou (2) remover as entidades persistidas depois de cada teste. Acreditamos que a primeira abordagem é mais adequada para este caso por dois motivos: (1) limpar o banco de dados é uma tarefa relativamente fácil; e (2) a segunda abordagem pode gerar uma dependência entre os testes, pois um teste pode vir a falhar caso outro teste não tenha removido adequadamente as entidades persistidas.

A Figura 23 apresenta a classe `NoDataTest` que verifica o sistema quando a camada de persistência está vazia. Para isso, a classe `NoDataTest` possui um *setup method*, chamado `configure`, que remove todas as eventuais entidades existentes no banco de dados.

```
@Singular
public class NoDataTest {

  private EventDao eventDao;
  private UserDao userDao;
  private EventUserDao dao;

  @Before
  public void configure() {
    eventDao = new EventDao();
    userDao = new UserDao();
    dao = new EventUserDao();
    eventDao.removeAll();
    userDao.removeAll();
    dao.removeAll();
  }

  @Test
  public void emptyData() {
    assertTrue(eventDao.list().isEmpty());
    assertTrue(userDao.list().isEmpty());
    assertTrue(dao.list().isEmpty());
  }

}
```

Figura 23. Classe `NoDataTest`.

Observa-se que o ponto de partida esperado para a classe `UserTest` é exatamente o *test fixture* da classe `NoDataTest`. Assim, caso a proposta não fosse utilizada e considerando que se deseje promover o reuso de código, existem, a princípio, duas possibilidades: (1) mover os testes da classe `UserTest` para a classe `NoDataTest`, já que o requisito dos testes da classe `UserTest` de remoção das entidades no banco de dados é feito na classe `NoDataTest`; ou (2) extrair o código do *setup method* da classe `NoDataTest` para uma classe auxiliar e promover o reuso de código através da estratégia *delegate setup*. A primeira abordagem limita a organização das classes de teste e impossibilita que o *setup method* da classe `UserTest` seja mantido e movido para a classe `NoDataTest`, pois, caso seja, irá causar uma falha indesejada no teste `emptyData`, da classe `NoDataTest`. Já a segunda abordagem obriga a criação de artefatos auxiliares de código e pode contribuir para testes de difícil compreensão. O *setup method* da classe `NoDataTest` deverá ser movido para um local diferente de onde foi originalmente definido, e isso pode dificultar a compreensão da relação de causa e efeito entre os *test fixtures* e as saídas do SUT.

A proposta deste trabalho pode ser utilizada para possibilitar que os *test fixtures* da classe `NoDataTest` sejam utilizados pelos testes da classe `UserTest` sem que seja necessário alterar as classes envolvidas. Através da anotação `@FixtureSetup`, indica-se ao *framework* de teste que os *test fixtures* da classe `NoDataTest` deverão ser utilizados pelos testes da classe `UserTest`. Com isso, promove-se o reuso de código com relativa facilidade. Destaca-se que o modelo de manipulação permite que o *test fixture* `userDao`, pertencente à classe `NoDataTest`, seja manipulado pelos testes da classe `UserTest`. Para isso, um atributo de mesmo nome deve ser declarado na classe `UserTest` e anotado com a anotação `@Fixture`. O *framework* de teste deverá injetar, na classe `UserTest`, o *test fixture* associado ao atributo `userDao` da classe `NoDataTest`.

A classe `EventTest`, mostrada na Figura 21, é análoga à classe `UserTest`. Assim, as mesmas observações feitas para esta última também podem ser aplicadas para a primeira. Esta classe também deverá indicar, através da anotação `@FixtureSetup`, que os *test fixtures* da classe `NoDataTest` serão utilizados por ela.

Os testes para a entidade associativa da classe `UserEvent` são definidos na classe `UserEventTest`, conforme mostrado na Figura 22, e dependem dos *test fixtures* `john` e `lecture`. Um desses *test fixtures* é

configurado na classe `UserTest`, enquanto o outro é configurado na classe `EventTest`. Assim, através do modelo de dependência, a classe `UserEventTest` pode utilizar os *test fixtures* das duas classes. Utiliza-se a anotação `@FixtureSetup` para definir as duas dependências.

É importante fazer uma ressalva a respeito da classe `NoDataTest`. Seu *test fixture* é singular, isto é, deve ser executado uma única vez durante um mesmo teste (ver Seção 4.1.5). Isso é indicado ao *framework* através da anotação `@Singular`. Destaca-se que a ausência dessa anotação causaria uma falha indesejada nos testes da classe `UserEventTest`. Sem a anotação, o *test fixture* de `NoDataTest` seria executado duas vezes: uma vez na execução dos *test fixtures* da classe `UserTest` e outra vez na execução dos *test fixtures* da classe `EventTest`. Nesse caso, a segunda execução removeria o *test fixture* `john` que fora persistido após a primeira execução. Através da anotação `@Singular` evita-se que o *framework* de teste realize a segunda execução, impedindo que o *test fixture* `john` seja removido.

6.2. EXPERIMENTO SOBRE REUSO DE CÓDIGO

Através do desenvolvimento de um sistema foi conduzido um experimento para avaliar o reuso de código obtido através da utilização da proposta. O sistema desenvolvido fez parte de um trabalho realizado por alunos da disciplina de Desenvolvimento Ágil de Sistemas do mestrado em Ciências da Computação da Universidade Federal de Santa Catarina. O trabalho consistiu no desenvolvimento, em grupo, de um sistema para o agendamento de horários para a realização de eventos com vários participantes. No desenvolvimento, os alunos deveriam utilizar um conjunto de práticas ágeis para o desenvolvimento do sistema, dentre as quais destaca-se o desenvolvimento de testes ao longo do projeto.

O experimento foi conduzido da seguinte maneira: inicialmente os testes foram desenvolvidos de forma iterativa, de acordo com as funcionalidades necessárias. Os testes foram criados através da linguagem Java e do *framework* JUnit e utilizaram-se das estratégias convencionais de *fixture setup*. Após o término do trabalho, um subconjunto dos testes foi escolhido arbitrariamente. Os testes escolhidos foram, então, reescritos manualmente para utilizar a proposta deste trabalho através do *framework* Story. O subconjunto dos testes para o experimento foi extraído de um conjunto total de 77 testes divididos em 11 classes de teste distintas. Esse subconjunto, chamado

doravante de **grupo de controle**, foi constituído por 24 testes, 4 classes de teste e uma classe auxiliar. O conjunto de testes reescritos manualmente a partir do grupo de controle será chamado de **grupo experimental**. O grupo experimental resultou em 24 testes, 14 classes de teste e uma classe auxiliar. A criação dos testes do grupo experimental levou em consideração as seguintes restrições: (1) para cada teste do grupo de controle deve existir um teste equivalente no grupo experimental; (2) a cobertura de código não deve ser diferente entre os dois grupos; (3) os conjuntos de *test fixtures* e de verificações para cada teste do grupo de controle deve ser o mesmo para o respectivo teste do grupo experimental; (4) nomes de variáveis, métodos e atributos existentes no grupo de controle e mantidos no grupo experimental devem ser preservados; (5) anotações do Story devem ser colocadas em uma linha individual; e (6) os testes podem ser reorganizados e reposicionados livremente, desde que as restrições anteriores sejam respeitadas.

Após a realização do experimento, foram realizadas as seguintes medições nos grupos de controle[5] e experimental[6]: (1) quantidade de linhas de código das classes de teste e classes auxiliares; (2) somatório da quantidade de linhas repetidas, excluindo-se asserções; (3) quantidade de repetições diferentes, excluindo-se asserções; (4) somatório da quantidade de linhas repetidas, incluindo-se asserções; e (5) quantidade de repetições diferentes, incluindo-se asserções.

Em todas as cinco medições realizadas foram ignoradas: linhas em branco, linhas de declaração de pacote e linhas de importação. Nas medições 2, 3, 4 e 5, anotações `@Test`, `@Before` e `@Fixture`, declarações de método idênticas e o símbolo de delimitação de bloco não contabilizaram repetições.

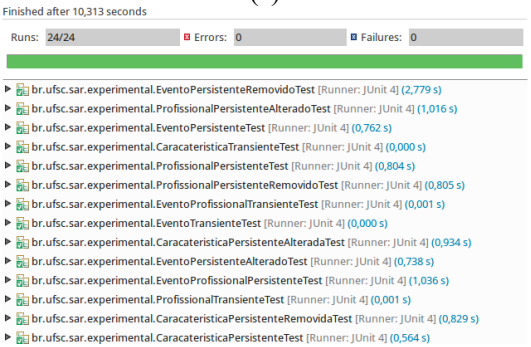
A Figura 24 apresenta os resultados para execuções de três agrupamentos distintos de testes, sendo esses: (a) testes do grupo de controle; (b) testes do grupo experimental; e (c) testes de ambos os grupos. Percebe-se, através da Figura 24 (a) e da Figura 24 (b), que o tempo de execução entre os testes dois grupos é consistentemente semelhante. Esse comportamento serve como indicativo razoável de que a terceira restrição não foi violada.

---

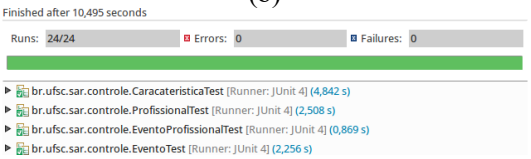
[5] https://github.com/lucasPereira/alocacaoDeHorarios/tree/testes-estoria/src/test/java/br/ufsc/sar/controle

[6] https://github.com/lucasPereira/alocacaoDeHorarios/tree/testes-estoria/src/test/java/br/ufsc/sar/experimental

(a)



(b)



(c)

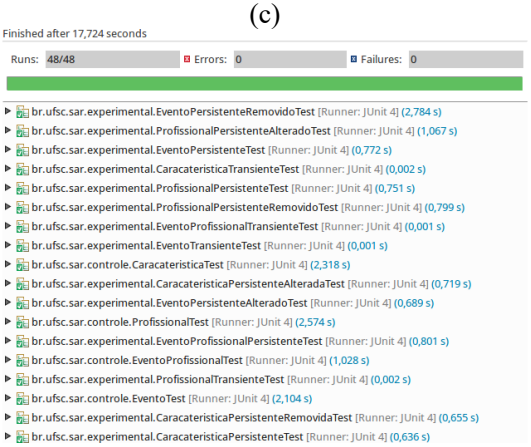


Figura 24. Execução do experimento sobre reuso de código.

O resultado do experimento pode ser observado através do gráfico da Figura 25. As barras pintadas em preto representam a medição das métricas para o grupo de controle, enquanto as barras pintadas em cinza representam a medição das métricas para o grupo experimental. Pode-se observar que o grupo experimental apresentou uma quantidade ligeiramente maior de linhas de código de teste. Esse

aumento pode ser justificado pela utilização das anotações necessárias para a utilização do *framework* Story. Apesar disso, o grupo experimental apresentou considerável redução no número de linhas repetidas, sendo um total de 126 para o grupo de controle e 66 para o grupo experimental. Levando-se em consideração a proporcionalidade de linhas de código de teste, o grupo de controle apresentou 40,91% de linhas repetidas, enquanto no grupo experimental esse número foi de 20,37%. Se comparado com o grupo de controle, observa-se também que o grupo experimental apresentou, considerando-se valores absolutos, uma redução de 47,62% das linhas de código repetidas.

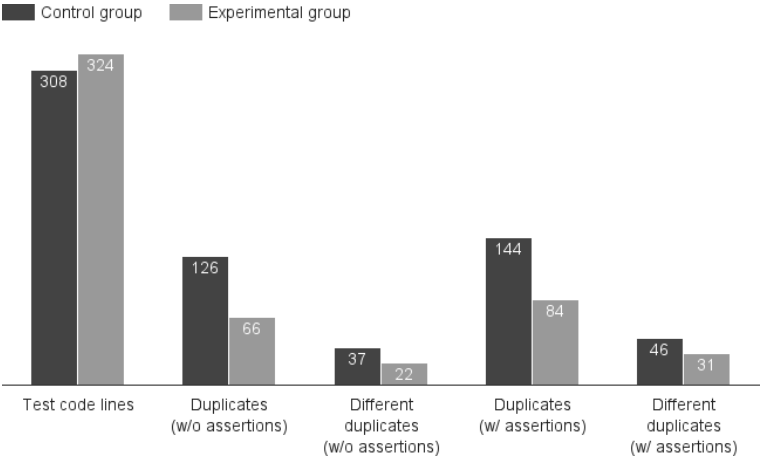


Figura 25. Resultado do experimento sobre reuso de código.

6.3. EXPERIMENTO SOBRE REUSO DE EXECUÇÃO

Neste experimento uma mesma suíte de teste foi executada duas vezes através do Story. Na primeira execução foi utilizada uma versão simplificada do Story, isto é, uma versão sem reuso de execução, enquanto na segunda execução foi utilizada a versão otimizada, isto é, com reuso de execução. O tempo para cada uma das execuções foi coletado com o objetivo de identificar possíveis diferenças quando a execução é realizada com ou sem reuso de execução.

Os testes utilizados no experimento foram extraídos do projeto de desenvolvimento do Sistema de Acompanhamento e Avaliações de Curso (SAAS). O SAAS é um sistema para o gerenciamento de questionários e respectivos resultados sobre os cursos da rede e-Tec Brasil (Cislaghi *et al.*, 2014). O sistema é desenvolvido através da

plataforma Java EE e seu desenvolvimento é guiado por testes (TDD). Para a realização do experimento foram escolhidos os testes do sistema que correspondem a testes de aceitação para a interface gráfica com o usuário. Uma vez que este tipo de teste é mais lento, isso facilita a identificação de possíveis diferenças entre executar os testes utilizando ou não reuso de execução de *test fixtures*. Os testes usam o Selenium, uma biblioteca para comunicar e interagir com o sistema através de navegadores Web (Alvestad, 2007).

Antes de executar os testes selecionados foi necessário reescrevê-los para que pudessem ser adaptados à proposta deste trabalho. Assim, os 32 testes selecionados resultaram em outros 32 testes adaptados para utilizar a proposta deste trabalho. Após a reescrita dos testes foram realizadas duas execuções diferentes conforme apresentado na Figura 26. Na execução simplificada, sem reuso de execução de *test fixtures*, os 32 testes levaram cerca de 1089 segundos para serem executados, enquanto na execução otimizada, com reuso de execução de *test fixtures*, os mesmos 32 testes levaram cerca de 131 segundos. Isso representou uma redução de aproximadamente 8 vezes no tempo de execução quando a execução otimizada foi utilizada.

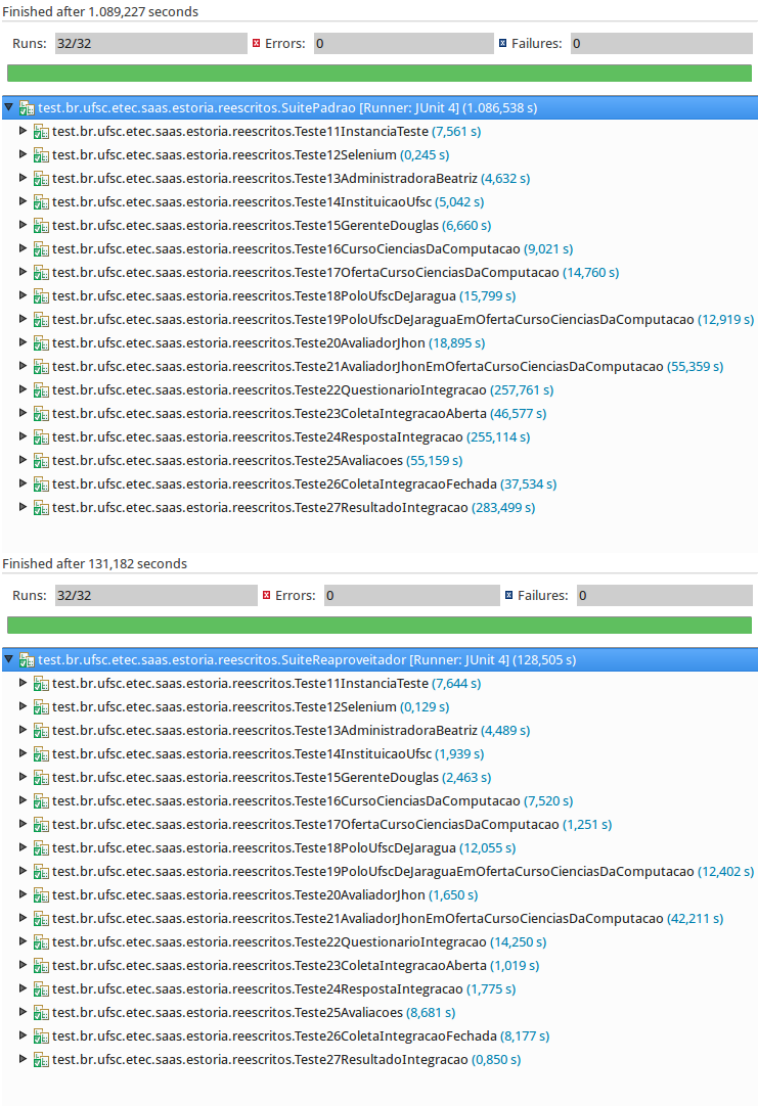


Figura 26. Execução do experimento sobre reuso de execução.

6.4. EXPERIMENTO SOBRE APRENDIZADO E UITILIZAÇÃO

Este experimento contou com a participação de Contagem 2 programadores e foi organizado de acordo com as seguintes etapas: (1) preparação; (2) aprendizado; (3) criação; (4) modificação; e (5) encerramento. Cada participante passou individualmente por cada uma das etapas. O objetivo deste experimento foi avaliar a estratégia proposta neste trabalho do pondo de vista do aprendizado e do uso.

6.4.1. Preparação

A preparação consistiu na apresentação de uma aplicação criada especificamente para a realização deste experimento. O domínio, os requisitos, a arquitetura e a implementação da aplicação foram apresentados aos participantes. Todas as demais etapas deste experimento tiveram como base a aplicação apresentada nesta etapa.

A aplicação, um sistema bancário, foi apresentada através de da explicação oral do domínio e dos requisitos da aplicação, de um diagrama de classes (Figura 27) e de um conjunto de classes Java (Apêndice I)[7]. Nesta etapa não foi buscado realizar nenhuma avaliação. Buscou-se, apenas, fornecer aos participantes uma base comum de conhecimento a respeito da aplicação.

[7] Também disponível em: https://github.com/lucasPereira/sistemaBancario/tree/experimento/java/br/ufsc/ine/leb/sistemaBancario

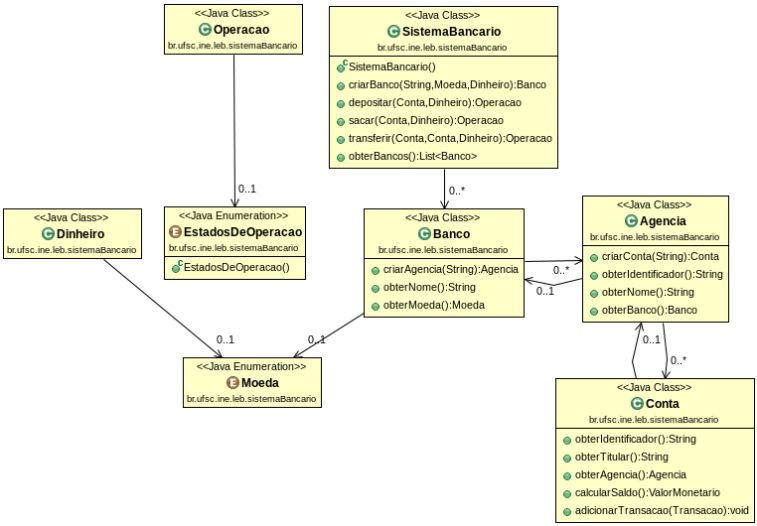


Figura 27. Diagrama de classes do Sistema Bancário.

6.4.2. Aprendizado

Na etapa de aprendizado o participante foi apresentado às estratégias convencionais de *fixture setup*, bem como à estratégia proposta. A apresentação de cada estratégia foi acompanhada de exemplos de uso da mesma (Apêndice II)[8]. Ao final da apresentação de cada estratégia, o participante realizou um exercício utilizando-se da respectiva estratégia apresentada. Um conjunto de testes incompletos (Apêndice III)[9] foi fornecido a cada participante. A tarefa do participante consistiu em incluir os *test fixtures* necessários para que os casos de teste passassem.

Esta etapa foi importante para avaliar o entendimento dos participantes em relação à proposta (primeiro ponto a ser esclarecido através dos experimentos). Foi observado se os participantes conseguiram compreender a estratégia proposta, bem como aplicá-la e utilizá-la corretamente em um caso de teste. Para isso, os seguintes

[8] Também disponível em: https://github.com/lucasPereira/sistemaBancario/tree/experimento/java/br/ufsc/ine/leb/sistemaBancario/experimento/etapa1

[9] Também disponível em: https://github.com/lucasPereira/sistemaBancario/tree/experimento/java/br/ufsc/ine/leb/sistemaBancario/experimento/etapa2

dados foram colhidos: (a) resultado do teste: se o teste passou ou não; (b) adequação da aplicação da estratégia: se a estratégia foi utilizada corretamente ou não; e (c) tempo de implementação: tempo necessário para o participante completar o caso de teste.

6.4.3. Criação

Na etapa de criação cada participante precisou implementar um conjunto de seis casos de teste (Apêndice IV) para cobrir os requisitos da aplicação. Os participantes puderam usar livremente toda e qualquer estratégia de *fixture setup* que lhe foi apresentada.

Nesta etapa pretendeu-se avaliar a estratégia proposta quanto ao uso (segundo ponto a ser esclarecido através dos experimentos). Objetivou-se, sobretudo, verificar se o participante vê utilidade prática na estratégia proposta neste trabalho e se, durante o desenvolvimento, o participante utiliza a estratégia por iniciativa própria. Para realizar a avaliação foram coletados os seguintes dados: (d) tempo de implementação dos casos de teste; (e) quantidade de classes de teste; (f) quantidade de `@FixtureSetup` declarados; (g) quantidade de `@Fixture` declarados; (h) quantidade de testes corretos passando; (i) quantidade de testes corretos falhando; e (j) quantidade de testes incorretos. Testes corretos são aqueles que verificam adequadamente o cenário descrito pelo respectivo caso de uso, enquanto testes incorretos não.

6.4.4. Modificação

Na etapa de modificação foi realizada uma mudança proposital na implementação da aplicação. A única mudança realizada consistiu na alteração da assinatura do método `criarBanco` da classe `SistemaBancario` (Apêndice V). Cada participante recebeu a tarefa de alterar os casos de teste desenvolvidos na etapa anterior de modo que refletissem as mudanças causadas na implementação.

Através desta etapa buscou-se avaliar a proposta quanto à manutenibilidade do código de teste (terceiro ponto a ser esclarecido através dos experimentos). Foram coletados os seguintes dados: (k) tempo de correção dos testes; (l) quantidade de testes corretos passando; (m) quantidade de testes corretos falhando; (n) quantidade de testes incorretos; e (o) quantidade de linhas modificadas.

6.4.5. Encerramento

Após a realização de todos os exercícios, foi aplicado um questionário (Apêndice VI) aos participantes com o objetivo de obter as seguintes informações: (p) experiência com desenvolvimento de sistemas; (q) experiência com desenvolvimento de testes; (r) estratégias de *fixture setup* conhecidas; (s) avaliação a respeito do esforço para aprendizado da estratégia proposta neste trabalho; (t) avaliação a respeito do esforço para utilização da estratégia proposta neste trabalho; (u) avaliação a respeito da utilidade prática da estratégia proposta neste trabalho; e (v) considerações, se houver, a respeito da estratégia proposta neste trabalho.

6.4.6. Resultados

Na etapa de **aprendizado**, todos os testes completados pelos participantes passaram. Além disso, os participantes conseguiram utilizar adequadamente todas as diferentes estratégias de *fixture setup*. Esse fator é importante, pois demonstra que os participantes do experimento conseguiram entender a estratégia proposta bem como utilizá-la corretamente. A Tabela 2 apresenta a relação da estratégia utilizada e o tempo que cada participante levou para completar os testes utilizando-se da respectiva estratégia. A última linha da tabela apresenta a média aritmética entre os tempos dos dois participantes.

	Inline setup	*Implicit setup*	*Delegate setup*	Estratégia proposta
Participante 1	00:05:20	00:02:05	00:05:05	00:04:50
Participante 2	00:08:00	00:05:30	00:07:05	00:06:40
Média	00:06:40	00:03:48	00:06:05	00:05:45

Tabela 2. Síntese dos dados coletados na etapa de aprendizado.

A média geral entre todas as estratégias foi de 00:05:34. Pode-se observar que não houve diferenças significativas entre as diferentes estratégias. Devido à pouca quantidade de dados, não se pode estabelecer nenhuma conclusão. Entretanto, alguns indicativos podem ser observados. Em ambos os casos o tempo para completar os testes com a estratégia proposta foi menor que com a estratégia *delegate setup*. Além disso, em ambos os casos a estratégia *implicit setup* se mostrou a mais rápida enquanto a estratégia *inline setup* a mais lenta.

A Tabela 3 apresenta uma síntese dos dados coletados na etapa de **criação** dos testes. Todos os testes implementados pelos participantes passaram e refletiram corretamente os casos de teste que deviam ser implementados. Por isso, os dados referentes a estas características foram omitidos na Tabela 3.

	Tempo de implementação dos casos de teste	Quantidade de classes de teste	Quantidade de `@FixtureSetup` declarados	Quantidade de `@Fixture` declarados
Participante 1	00:41:00	3	2	9
Participante 2	00:52:35	6	5	10

Tabela 3. Síntese dos dados coletados na etapa de criação.

Pode-se observar através da Tabela 3 que ambos participantes utilizaram a estratégia proposta neste trabalho. Essa característica é importante, pois indica que os participantes viram utilidade prática na estratégia proposta. Destaca-se o fato de que o Participante 2 utilizou mais vezes a anotação `@FixtureSetup`. Isso justifica o fato de ter utilizado mais classes de teste que o Participante 1. A estratégia proposta é altamente dependente da organização das classes de teste.

Na etapa de **modificação**, a mudança causada na implementação da aplicação teve como objetivo investigar a robustez dos testes desenvolvidos. Para isso, foi modificada assinatura do método responsável pela criação de objetos da classe `Banco`, que é uma classe necessária para a implementação de todos os seis casos de teste apresentados aos participantes.

A Tabela 4 apresenta a síntese dos dados coletados na etapa de modificação. Destaca-se que após os participantes realizarem as modificações necessárias nos testes, todos os testes continuaram passando e refletindo corretamente os casos de teste. Por isso, os dados referentes a estas características foram omitidos na Tabela 4.

	Tempo de correção dos testes	Quantidade de linhas modificadas
Participante 1	00:00:40	1
Participante 2	00:00:35	1

Tabela 4. Síntese dos dados coletados na etapa de modificação.

A Tabela 4 mostra que apesar da alteração na implementação da aplicação afetar os *test fixtures* de todos os seis testes, o esforço de manutenção foi consideravelmente baixo. Ambos participantes levaram

poucos segundos para identificar e corrigir um problema que afetou todos os testes. Pode-se observar que em ambos os casos foi necessário modificar apenas uma linha para fazer com que os testes voltassem a passar. Isso indica que os participantes evitaram a duplicação de código entre os testes. Uma análise no código produzido por cada participante mostrou que ambos utilizaram a estratégia proposta neste trabalho para definir o *test fixture* referente a classe `Banco`.

Na etapa de **encerramento** os participantes responderam o questionário apresentado no Apêndice VI. A Tabela 5 apresenta a experiência que cada participante julgou ter em relação ao desenvolvimento de sistemas e desenvolvimento de testes. O Participante 1 considerou ter experiência moderada tanto no desenvolvimento de sistemas quanto no desenvolvimento de testes. Já o Participante 2 considerou ter baixa experiência no desenvolvimento de sistemas e nenhuma experiência com desenvolvimento de testes.

	Experiência com desenvolvimento de sistemas	Experiência com desenvolvimento de testes
Participante 1	Média	Média
Participante 2	Baixa	Nenhuma

Tabela 5. Experiência de desenvolvimento dos participantes.

Os participantes também foram perguntados sobre quais estratégias de *fixture setup* já conheciam antes da realização do experimento. O Participante 1 afirmou conhecer todas as três estratégias já existentes apresentadas, *inline setup*, *implicit setup* e *delegate setup*. Já o Participante 2 afirmou conhecer apenas as estratégias *inline setup* e *implicit setup*. O Participante 2 foi questionado sobre os motivos de ter declarado não ter nenhuma experiência com desenvolvimento de testes sendo que já conhecia as duas estratégias de *fixture setup*. O participante respondeu que tinha conhecimento das estratégias por ter aprendido sobre elas durante as aulas da graduação, mas que nunca desenvolveu testes em um projeto real de desenvolvimento.

Os participantes também foram perguntados sobre o esforço de aprendizado, o esforço de utilização e se viram utilidade prática na estratégia proposta neste trabalho. A Tabela 6 apresenta a síntese dessas informações.

	Esforço de aprendizado	Esforço de utilização	Utilidade prática
Participante 1	Baixo	Baixo	Útil
Participante 2	Baixo	Baixo	Útil

Tabela 6. Avaliação dos participantes sobre a estratégia proposta.

Ambos participantes consideraram que o tanto o esforço de aprendizado quanto o esforço da utilização da estratégia proposta neste trabalho foram baixos. Estas informações corroboram com os tempos apresentados na Tabela 2 em relação à etapa de aprendizado, onde não se percebeu diferenças significativas entre os tempos da estratégia proposta em relação às demais estratégias.

Os dois participantes também concordaram ao dizer que a estratégia proposta tem utilidade prática em algumas situações. Estas informações corroboram os dados apresentados na Tabela 3, uma vez que, conforme mostrado na tabela, ambos participantes utilizaram a estratégia proposta para a implementação dos casos de teste.

6.5. DISCUSSÃO

Através deste capítulo foi mostrado um exemplo de uso e experimentos para avaliar a proposta deste trabalho. O exemplo de uso (ver Seção 6.1) foi realizado a partir da adaptação de testes de um sistema de escalonamento de horários de eventos desenvolvido em uma disciplina de desenvolvimento ágil de sistemas. O exemplo de uso serve como indício de que a proposta deste trabalho pode ser aplicada para testes de um ambiente real de desenvolvimento. Esse indício é reforçado pelo experimento sobre aprendizado e utilização (ver Seção 6.4) onde a proposta deste trabalho foi utilizada por dois programadores.

Através do experimento sobre reuso de código (ver Seção 6.2) e do experimento sobre reuso de execução (ver Seção 6.3) foi possível avaliar comparativamente a proposta deste trabalho em relação às estratégias de *fixture setup* já existentes. Os experimentos mostraram que, para o contexto em que foram aplicados, foi possível atingir uma redução de 47,62% das linhas de código de código de teste e uma redução de 8 vezes no tempo de execução dos testes.

Assim, o exemplo de uso e os experimentos apresentados neste capítulo sugerem que a proposta deste trabalho pode ser utilizada na prática em um ambiente real de desenvolvimento de modo que contribua tanto com o reuso de código quanto com o reuso de execução.

6.5.1. Ameaças à Validade

Os testes adaptados para o exemplo de uso, os testes do grupo experimental do experimento sobre reuso de código e os testes reescritos do experimento sobre reuso de execução foram todos desenvolvidos pelo próprio autor deste trabalho. Isso representa uma ameaça à validade, uma vez que o conhecimento prévio do autor em relação à proposta pode ter influenciado a criação dos testes.

O experimento sobre reuso de execução apresenta, ainda, uma outra ameaça à validade. A seleção dos testes para a realização do experimento não foi aleatória. Foram selecionados, propositalmente, testes de aceitação que possuem entre si uma relação naturalmente sequencial de uso do SUT. Além disso, o experimento registrou o tempo de execução para cada suíte de teste apenas uma vez. Esta ameaça poderia ter sido eliminada caso os tempos de execuções tivessem sido registrados em todas as vezes que o experimento foi repetido, permitindo, assim, que fosse realizada uma média aritmética entre esses tempos.

Diferentemente dos outros experimentos, o experimento sobre aprendizado e utilização não apresenta a ameaça da validade de que a criação dos testes pode ter sofrido influência do autor desta proposta. Entretanto, a combinação de dois fatores pode ter influenciado nos resultados obtidos na etapa de aprendizado (ver Seção 6.4.2): (1) os casos de teste foram os mesmos para todas as estratégias de *fixture setup*; e (2) foi definida uma ordenação entre as estratégias de *fixture setup* utilizadas para criar os testes. A ordem de utilização das estratégias foi: *inline setup*, *implicit setup*, *delegate setup* e a estratégia proposta neste trabalho. Como os casos de teste foram os mesmos para todas as estratégias, então o conhecimento adquirido ao implementar o teste com a estratégia anterior pode ter facilitado a implementação do teste com as estratégias seguintes. Além dessa questão, também pode-se citar a pouca quantidade de participantes como uma ameaça à validade.

7. CONCLUSÃO

Este trabalho abordou o reuso de *test fixtures*. Cada teste deve, antes de verificar o comportamento esperado, colocar o SUT em um estado de interesse. Esse estado de interesse corresponde às precondições necessárias para o teste. As verificações realizadas pelo teste partem do pressuposto que as precondições necessárias foram satisfeitas e que o SUT está no estado de interesse desejado. A tarefa de colocar o SUT no estado de interesse desejado é realizada através da execução de *test fixtures*. Portanto, *test fixtures* fazem parte, não do código de produção, mas sim, do código de teste.

O sucesso a longo prazo da atividade de automação de testes é fortemente influenciado pela manutenibilidade dos testes (Greiler *et al.*, 2013). Meszaros (2006) defende que para se alcançar uma boa manutenibilidade é necessário que sejam satisfeitas algumas características: (1) devem ser fáceis de executar; (2) devem ser fáceis de ler e escrever; e (3) devem ser robustos. Este trabalho mostrou que promover o reuso de *test fixtures* de forma adequada pode contribuir positivamente com estas características. A proposta deste trabalho contribui para que testes sejam mais robustos, pois promove o reuso de código de *test fixture*. Contribui também para que os testes sejam mais fáceis de ler e escrever, pois permite que classes de teste utilizem *test fixtures* já existentes em outras classes de teste sem que seja necessário alterar as estruturas das classes envolvidas. Por fim, contribui para que os testes sejam mais fáceis de executar, pois promove o reuso de execução de *test fixture*.

As estratégias de *fixture setup* são importantes, pois têm impacto direto nas características desejadas para o teste, mencionadas anteriormente. Cada estratégia de *fixture setup* afeta de forma diferente as características desejadas. Além disso, não existe um guia padrão para utilização das estratégias de *fixture setup*. Uma estratégia pode ter impacto diferente dependendo do contexto em que for utilizada. Por exemplo, a estratégia *inline setup* facilita a compreensão da relação de causa e efeito entre os *test fixtures* e as saídas do SUT, porém também causa duplicação de código de teste. Assim, a decisão de utilizar esta estratégia deve ponderar se o resultante dos efeitos que a estratégia causa nas características desejadas é positivo ou não. Cada estratégia de *fixture setup* irá apresentar vantagens e desvantagens que variam conforme o contexto em que são utilizadas. Por isso, Meszaros (2006) defende que o melhor caminho para o desenvolvimento de testes com boa manutenibilidade é buscar um equilíbrio entre as estratégias de

fixture setup, de modo que ao escolher uma estratégia seja levado em consideração o efeito global que a estratégia escolhida terá na manutenibilidade do código de teste.

A proposta deste trabalho possibilita a implementação de uma nova estratégia de *fixture setup*. Assim como ocorre com as estratégias já existentes, a nova estratégia pode ser mais ou menos adequada dependendo do contexto em que for utilizada. Pretendeu-se com este trabalho criar de uma estratégia que preencha o espaço não ocupado pelas estratégias convencionais de forma que uma mesma estratégia possa satisfazer os seguintes requisitos: (1) promover o reuso de código de *test fixture*; (2) promover o reuso de execução de *test fixture*; e (3) permitir que classes de teste utilizem *test fixtures* de uma ou mais classes de teste já existentes sem que, para isso, seja necessário alterar a estrutura das classes envolvidas. Em nenhuma das estratégias convencionais estes itens são contemplados simultaneamente. No *implicit setup* e no *delegate setup* apenas o primeiro item é contemplado. Já no *inline setup*, nenhum dos itens são contemplados.

Este trabalho apresentou modelos que permitem reusar *test fixtures* entre classes de teste. Os modelos apresentados neste trabalho permitem que classes de teste sejam construídas de um modo naturalmente hierárquico. O desenvolvimento de uma classe de teste pode ter como ponto de partida os *test fixtures* de uma classe de teste já existente. Através dos modelos deste trabalho é possível criar uma hierarquia múltipla de classes de teste, onde em cada classe de teste da hierarquia novos *test fixtures* são definidos com base na classe anterior.

7.1. CONTRIBUIÇÕES

Este trabalho abrangeu tanto o reuso de código quanto o reuso de execução através de uma mesma estratégia de *fixture setup*. Através da proposta foi possível atingir, no contexto dos experimentos em que foi aplicada, uma redução de 47,62% das linhas de código de código de teste e uma redução de 8 vezes no tempo de execução dos testes.

Este trabalho também resultou na construção na construção do *framework* Story onde os modelos propostos foram implementados. Além de dar suporte a uma nova estratégia de *fixture setup* que permite promover tanto o reuso de código quanto o reuso de execução, o Story também suporta a execução de testes escritos para o *framework* JUnit. Todo o código fonte do Story foi disponibilizado de forma aberta através da plataforma GitHub. Com isso, espera-se que o Story possa ser

utilizado em projetos de desenvolvimento de *software* e possa, também, contribuir com o desenvolvimento de trabalhos futuros.

A proposta apresentada neste trabalho resultou na publicação do artigo intitulado *Execution and Code Reuse between Test Classes* no SERA 2016[10] (Silva e Vilain, 2016).

7.2. COMPARAÇÃO COM TRABALHOS RELACIONADOS

O trabalho de Christensen *et al.* (2006) é o que mais se assemelha à proposta deste trabalho. Os autores propõem uma extensão a um *framework* de teste com o objetivo de promover o reuso de *test fixtures* de testes para o banco de dados. Assim como neste trabalho, a abordagem de Christensen *et al.* (2006) também utilizou um modelo de dependência. Entretanto, no trabalho correlato os *test fixtures* podem ser apenas executados, mas não manuseados, por testes de outras classes. Com isso, somente é possível reusar *test fixtures* de uma outra classe de teste caso os *test fixtures* forem salvos através de um mecanismo de persistência. Além disso, para cada teste em que um *test fixture* é inserido na camada de persistência, deve-se definir um teste de remoção onde o respectivo *test fixture* deve ser removido. Isso é feito para evitar falhas indesejadas entre os testes. Outra limitação do trabalho é que os testes devem, obrigatoriamente, seguir uma ordem de execução, violando, assim, o princípio da independência entre testes (Meszaros *et al.*, 2003).

O trabalho proposto por Mugridge e Cunningham (2005) também aborda o reuso de *test fixtures*. Entretanto, a abordagem proposta, diferentemente da abordagem deste trabalho, é aplicada ao reuso de tabelas Fit e não ao reuso de *test fixtures* para código de teste. Os *test fixtures* reusáveis devem ser definidos *a priori* para que, posteriormente, sejam utilizados no desenvolvimento dos testes. Essa limitação pode dificultar a criação dos testes, uma vez que nem sempre é possível saber antecipadamente quais serão os *test fixtures* necessários. Além disso, os autores propõem que o conjunto de testes, conectados através das tabelas Fit reusáveis, sejam executados como um único grande teste. Isso difere da abordagem adotada neste trabalho, onde, apesar de existir a dependência entre classes de teste, cada teste é independente e pode ser executado individualmente.

[10] 14th International Conference on Software Engineering Research, Management and Applications.

Na proposta defendida por Longo *et al.* (2015) o reuso de *test fixtures* é promovido através da definição destes em um repositório externo centralizado. Os *test fixtures* são, então, injetados, pela ferramenta Picon, nas classes de teste. A utilização de nomes comuns para os *test fixtures* é adotada tanto no trabalho proposto por Longo *et al.* (2015) quanto nesta proposta. Uma diferença entre esta proposta e o trabalho apresentado por Longo *et al.* (2015) é que neste último os *test fixtures* devem ser definidos através de uma linguagem específica, diferente da linguagem utilizada para a codificação dos testes. Além disso, o trabalho não permite que testes utilizem *test fixtures* de outras classes de teste.

7.3. TRABALHOS FUTUROS

Como trabalhos futuros espera-se que possam ser desenvolvidos métodos para identificar automaticamente quais *test fixtures* podem ser coletivizados entre os testes. Dessa forma, para promover o reuso de execução, não seria mais necessário definir explicitamente quais testes são seguros e quais não são. Através de uma identificação automática, o próprio *framework* de teste poderia identificar quais testes são seguros e quais são inseguros. Também se espera que futuramente a proposta possa ser utilizada em um projeto real de desenvolvimento de forma que os seus impactos na manutenibilidade dos testes possam ser avaliados a partir de uma perspectiva de utilização a longo prazo.

REFERÊNCIAS

ALVESTAD, K. 2007. **Domain Specific Languages for Executables Specifications**. Institutt for datateknikk og informasjonsvitenskap.

BECK, K. 2002. **Test Driven Development: By Example**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

BECK, K. 2000. **Extreme programming explained: embrace change**. Addison-Wesley Professional.

BEIZER, B. 1990. **Software Testing Techniques (2nd ed.)**. Van Nostrand Reinhold Co., New York, NY, USA.

BENEDEN, P. J. 2009. **Animal parasites and messmates**. BiblioBazaar, LLC.

BERNER, S., WEBER, R., e Keller, R. K. 2005. **Observations and lessons learned from automated testing**. In Proceedings of the 27th International Conference on Software engineering (ICSE '05). ACM, New York, NY, USA, 571-579. DOI=http://dx.doi.org/10.1145/1062455.1062556.

BERTOLINO, A. 2007. **Software Testing Research: Achievements, Challenges, Dreams**. In Procedings of the 2007 Future of Software Engineering (FOSE '07). IEEE Computer Society, Washington, DC, USA, 85-103. DOI=http://dx.doi.org/10.1109/FOSE.2007.25.

BORG, R., e KROPP, M. 2011. **Automated acceptance test refactoring**. In Proceedings of the 4th Workshop on Refactoring Tools (WRT '11). ACM, New York, NY, USA, 15-21. DOI=http://dx.doi.org/10.1145/1984732.1984736.

BOURQUE, P., e FAIRLEY, R. E. 2014. **Guide to the Software Engineering Body of Knowledge (Swebok®): Version 3.0 (3rd ed.)**. IEEE Computer Society Press, Los Alamitos, CA, USA.

CANFORA, G., CIMITILE, A., GARCIA, F., PIATTINI, M., e VISAGGIO, C. A. 2006. **Evaluating advantages of test driven development: a controlled experiment with professionals**. In Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06). ACM, New York, NY, USA, 364-371. DOI=http://dx.doi.org/10.1145/1159733.1159788.

CHRISTENSEN, C. A., GUNDERSBORG, S., DE LINDE, K., e TORP, K. 2006. **A unit-test framework for database applications**. In

10th International Database Engineering and Applications Symposium (IDEAS'06). IEEE, 11-20.

CISLAGHI, R., WILGES, B., NASSAR, S. M., HIURA, D. L., e MATEUS, G. P. 2014. **Avaliação de polos sob uma perspectiva georreferenciada**. Proceedings of the 11th Congresso Brasileiro de Ensino Superior a Distância (ESUD' 14), Florianópolis, 771-781.

ERDOGMUS, H., MORISIO, M., e TORCHIANO, M. 2005. **On the Effectiveness of the Test-First Approach to Programming**. IEEE Transactions on Software Engineering 31, 3 (March 2005), 226-237. DOI=http://dx.doi.org/10.1109/TSE.2005.37.

EMERY, D. H. 2009. **Writing Maintainable Automated Acceptance Tests**. In Agile Testing Workshop, Agile Development Practices. Orlando, Florida.

FREEMAN, S., e PRYCE, N. 2009. **Growing Object-Oriented Software, Guided by Tests (1st ed.)**. Addison-Wesley Professional.

GREILER, M., DEURSEN, A., e STOREY, M. A. 2013. **Automated Detection of Test Fixture Strategies and Smells**. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST '13). IEEE Computer Society, Washington, DC, USA, 322-331. DOI=http://dx.doi.org/10.1109/ICST.2013.45.

GREILER, M., ZAIDMAN, A., DEURSEN, A., e STOREY, M. A. 2013. **Strategies for avoiding text fixture smells during software evolution**. In Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13). IEEE Press, Piscataway, NJ, USA, 387-396.

HANSSEN, G. K., e HAUGSET, B. 2009. **Automated Acceptance Testing Using Fit**. In Proceedings of the 42nd Hawaii International Conference System Sciences (HICSS '09). IEEE Computer Society, 1-8. DOI=http://dx.doi.org/10.1109/HICSS.2009.83.

KAMALRUDIN, M., SIDEK, S., AIZA, M. N., e ROBINSON, M. 2013. **Automated Acceptance Testing Tools Evaluation. In Agile Software Development**. Sci. Int, 4, 1053-1058.

LONGO, D. H., WILGES, B., VILAIN, P., and CISLAGHI, R. 2015. **Fixture Setup through Object Notation for Implicit Test Fixtures**. Journal of Computer Science 11, 6, 794.

MATTSSON, M., BOSCH, J., e FAYAD, M. E. 1999. **Framework integration problems, causes, solutions**. Commun. ACM 42, 10 (October 1999), 80-87. DOI=http://dx.doi.org/10.1145/317665.317679.

MUGRIDGE, R., e CUNNINGHAM, W. 2005. **Agile test composition**. In Extreme Programming and Agile Processes in Software Engineering. Springer Berlin Heidelberg, 137-144.

MUGRIDGE, R., e CUNNINGHAM, W. 2005. **Fit for developing software: framework for integrated tests**. Pearson Education.

MESZAROS. G. 2006. **xUnit Test Patterns: Refactoring Test Code**. Prentice Hall PTR, Upper Saddle River, NJ, USA.

MESZAROS, G., SMITH, S., e ANDREA, J. 2003. **The test automation manifesto**. In Proceedings of the 3rd XP Universe Conference (XP'03). New Orleans, LA.

PINTO, L. S., SINHA, S., e ORSO, A. 2012. **Understanding myths and realities of test-suite evolution**. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12). ACM, New York, NY, USA, Article 33, 11 pages. DOI=http://dx.doi.org/10.1145/2393596.2393634.

ROMBACH, V. R. B. D., KITCHENHAM, K. S. B., e SELBY, D. P. R. W. 2007. **Empirical Software Engineering Issues**.

SILVA, L. P., e VILAIN, P. 2016. **Execution and Code Reuse between Test Classes**. In Proceedings of the 14th International Conference on Software Engineering Research, Management and Applications (SERA 2016).

SOBERNIG, S., e ZDUN, U. 2010. **Inversion-of-control layer**. In Proceedings of the 15th European Conference on Pattern Languages of Programs (EuroPLoP '10). ACM, New York, NY, USA, Article 21, 22 pages. DOI=http://dx.doi.org/10.1145/2328909.2328935.

SWEET, R. E. 1985. **The Mesa programming environment**. In Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments (SLIPE '85), James Purtilo (Ed.). ACM, New York, NY, USA, 216-229. DOI=http://dx.doi.org/10.1145/800225.806843.

TIWARI, R., e GOEL, N. 2013. **Reuse: reducing test effort**. SIGSOFT Softw. Eng. Notes 38, 2 (March 2013), 1-11. DOI=http://dx.doi.org/10.1145/2439976.2439982.

TSAI, W. T., SAIMI, A., YU, L., e PAUL, R. 2003. **Scenario-based object-oriented testing framework**. In Quality Software, 2003. Proceedings. Third International Conference on (pp. 410-417). IEEE.

APÊNDICE I – APLICAÇÃO SISTEMA BANCÁRIO

```java
package br.ufsc.ine.leb.sistemaBancario;

import java.util.LinkedList;
import java.util.List;

public class Agencia {

	private Banco banco;
	private String nome;
	private Integer codigo;
	private List<Conta> contas;

	protected Agencia(String nome, Integer codigo, Banco banco) {
		this.nome = nome;
		this.codigo = codigo;
		this.banco = banco;
		this.contas = new LinkedList<>();
	}

	public Conta criarConta(String titular) {
		Conta conta = new Conta(titular, contas.size() + 1, this);
		contas.add(conta);
		return conta;
	}

	public String obterIdentificador() {
		return String.format("%03d", codigo);
	}

	public String obterNome() {
		return nome;
	}

	public Banco obterBanco() {
		return banco;
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario;

import java.util.LinkedList;
import java.util.List;

public class Banco {

	private Moeda moeda;
	private List<Agencia> agencias;
	private String nome;

	protected Banco(String nome, Moeda moeda, Dinheiro taxa) {
		this.nome = nome;
		this.moeda = moeda;
		this.agencias = new LinkedList<>();
	}

	public Agencia criarAgencia(String nome) {
		Agencia agencia = new Agencia(nome, agencias.size() + 1, this);
		agencias.add(agencia);
		return agencia;
	}

	public String obterNome() {
		return nome;
	}

	public Moeda obterMoeda() {
		return moeda;
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario;

import java.util.LinkedList;
import java.util.List;

public class Conta {

	private String titular;
	private Integer codigo;
	private Agencia agencia;
	private List<Transacao> transacoes;

	protected Conta(String titular, Integer codigo, Agencia agencia) {
		this.titular = titular;
		this.codigo = codigo;
		this.agencia = agencia;
		this.transacoes = new LinkedList<>();
	}

	public String obterIdentificador() {
		return String.format("%04d-%d", codigo, titular.length() % 10);
	}

	public String obterTitular() {
		return titular;
	}

	public Agencia obterAgencia() {
		return agencia;
	}

	public ValorMonetario calcularSaldo() {
		Moeda moeda = obterAgencia().obterBanco().obterMoeda();
		ValorMonetario saldo = new Dinheiro(moeda, 0, 0).positivo();
		for (Transacao transacao : transacoes) {
			saldo = transacao.contabilizar(saldo);
		}
		return saldo;
	}

	public void adicionarTransacao(Transacao transacao) {
		transacoes.add(transacao);
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario;

public class Dinheiro {

	private Moeda moeda;
	private Integer inteiro;
	private Integer fracionado;

	public Dinheiro(Moeda moeda, Integer inteiro, Integer fracionado) {
		this.moeda = moeda;
		this.inteiro = inteiro;
		this.fracionado = fracionado;
		normalizar();
	}

	private void normalizar() {
		Integer soma = obterQuantiaEmEscala();
		Integer baseFracionaria = moeda.obterBaseFracionaria();
		this.inteiro = (soma - (soma % baseFracionaria)) / baseFracionaria;
		this.fracionado = soma % baseFracionaria;
	}

	public Integer obterQuantiaEmEscala() {
		return Math.abs(inteiro) * moeda.obterBaseFracionaria() + Math.abs(fracionado);
	}

	public Moeda obterMoeda() {
		return moeda;
	}

	public String formatado() {
		return zero() ? formatarSemMoeda() : formatarComMoeda();
	}

	private String formatarSemMoeda() {
		return String.format("%d,%02d", inteiro, fracionado);
	}

```

```java

	private String formatarComMoeda() {
		return String.format("%d,%02d %s", inteiro, fracionado, moeda.toString());
	}

	private Boolean zero() {
		return inteiro == 0 && fracionado == 0;
	}

	public ValorMonetario positivo() {
		return new ValorMonetario(moeda).somar(this);
	}

	public ValorMonetario negativo() {
		return new ValorMonetario(moeda).subtrair(this);
	}

	@Override
	public boolean equals(Object objeto) {
		if (objeto instanceof Dinheiro) {
			Dinheiro outro = (Dinheiro) objeto;
			Boolean mesmaMoeda = moeda.equals(outro.moeda);
			Boolean mesmoValor = inteiro.equals(outro.inteiro) && fracionado.equals(outro.fracionado);
			return zero() || (mesmoValor && mesmaMoeda);
		}
		return super.equals(objeto);
	}

	@Override
	public String toString() {
		return formatado();
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario;

public class Entrada implements Transacao {

	private Dinheiro quantia;

	public Entrada(Conta conta, Dinheiro quantia) {
		this.quantia = quantia;
	}

	@Override
	public ValorMonetario obterValorMonetario() {
		return quantia.positivo();
	}

	@Override
	public ValorMonetario contabilizar(ValorMonetario saldo) {
		return saldo.somar(quantia);
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario;

public enum EstadosDeOperacao {

	SUCESSO, SALDO_INSUFICIENTE, MOEDA_INVALIDA

}
```

```java
package br.ufsc.ine.leb.sistemaBancario;

public enum Moeda {

	BRL("R$", 100), USD("$", 100), CHF("Fr", 100);

	private String simbolo;
	private Integer baseFracionaria;

	private Moeda(String simbolo, Integer escala) {
		this.simbolo = simbolo;
		this.baseFracionaria = escala;
	}

	public String obterSimbolo() {
		return simbolo;
	}

	public Integer obterBaseFracionaria() {
		return baseFracionaria;
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario;

public class Operacao {

	private EstadosDeOperacao estado;

	public Operacao(EstadosDeOperacao estado, Transacao... transacoes) {
		this.estado = estado;
	}

	public EstadosDeOperacao obterEstado() {
		return estado;
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario;

public class Saida implements Transacao {

	private Dinheiro quantia;

	public Saida(Conta conta, Dinheiro quantia) {
		this.quantia = quantia;
	}

	@Override
	public ValorMonetario obterValorMonetario() {
		return quantia.negativo();
	}

	@Override
	public ValorMonetario contabilizar(ValorMonetario saldo) {
		return saldo.subtrair(quantia);
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario;

import java.util.LinkedList;
import java.util.List;

public class SistemaBancario {

	private List<Banco> bancos;

	public SistemaBancario() {
		bancos = new LinkedList<>();
	}

	public Banco criarBanco(String nome, Moeda moeda) {
		Banco banco = new Banco(nome, moeda, new Dinheiro(moeda, 0, 0));
		bancos.add(banco);
		return banco;
	}

	public Operacao depositar(Conta conta, Dinheiro quantia) {
		Transacao entrada = new Entrada(conta, quantia);
		EstadosDeOperacao estado = EstadosDeOperacao.SUCESSO;
		if (moedaInvalida(conta, quantia)) {
			entrada = new TransacaoNaoRealizada(entrada);
			estado = EstadosDeOperacao.MOEDA_INVALIDA;
		}
		conta.adicionarTransacao(entrada);
		return new Operacao(estado, entrada);
	}

	public Operacao sacar(Conta conta, Dinheiro quantia) {
		ValorMonetario saldo = conta.calcularSaldo();
		Transacao saida = new Saida(conta, quantia);
		EstadosDeOperacao estado = EstadosDeOperacao.SUCESSO;
		if (saldo.negativo() || saldoFicaraNegativo(saldo, quantia)) {
			saida = new TransacaoNaoRealizada(saida);
			estado = EstadosDeOperacao.SALDO_INSUFICIENTE;
		}
		if (moedaInvalida(conta, quantia)) {
			saida = new TransacaoNaoRealizada(saida);
			estado = EstadosDeOperacao.MOEDA_INVALIDA;
		}
		conta.adicionarTransacao(saida);
		return new Operacao(estado, saida);
	}
```

```
46
47          public Operacao transferir(Conta origem, Conta destino, Dinheiro quantia) {
48                  Transacao saida = new Saida(origem, quantia);
49                  Transacao entrada = new Entrada(destino, quantia);
50                  EstadosDeOperacao estado = EstadosDeOperacao.SUCESSO;
51                  if (moedaInvalida(origem, quantia) || moedaInvalida(destino, quantia)) {
52                          saida = new TransacaoNaoRealizada(saida);
53                          entrada = new TransacaoNaoRealizada(entrada);
54                          estado = EstadosDeOperacao.MOEDA_INVALIDA;
55                  }
56                  origem.adicionarTransacao(saida);
57                  destino.adicionarTransacao(entrada);
58                  return new Operacao(estado, saida, entrada);
59          }
60
61          private Boolean saldoFicaraNegativo(ValorMonetario saldo, Dinheiro quantia) {
62                  return saldo.obterQuantia().obterQuantiaEmEscala() < quantia.obterQuantiaEmEscala();
63          }
64
65          private Boolean moedaInvalida(Conta conta, Dinheiro quantia) {
66                  Moeda moedaDoBanco = conta.obterAgencia().obterBanco().obterMoeda();
67                  Moeda moedaDaOperacao = quantia.obterMoeda();
68                  return !moedaDoBanco.equals(moedaDaOperacao);
69          }
70
71          public List<Banco> obterBancos() {
72                  return bancos;
73          }
74
75  }
```

```
1   package br.ufsc.ine.leb.sistemaBancario;
2
3   public interface Transacao {
4
5           public ValorMonetario obterValorMonetario();
6
7           public ValorMonetario contabilizar(ValorMonetario saldo);
8
9   }
```

```
1   package br.ufsc.ine.leb.sistemaBancario;
2
3   public class TransacaoNaoRealizada implements Transacao {
4
5           private Transacao transacao;
6
7           public TransacaoNaoRealizada(Transacao transacao) {
8                   this.transacao = transacao;
9           }
10
11          @Override
12          public ValorMonetario obterValorMonetario() {
13                  return transacao.obterValorMonetario();
14          }
15
16          @Override
17          public ValorMonetario contabilizar(ValorMonetario saldo) {
18                  return saldo;
19          }
20
21  }
```

```java
package br.ufsc.ine.leb.sistemaBancario;

public class ValorMonetario {

	private Moeda moeda;
	private Integer sinal;
	private Dinheiro quantia;

	public ValorMonetario(Moeda moeda) {
		this(moeda, 0);
	}

	private ValorMonetario(Moeda moeda, Integer valor) {
		this.moeda = moeda;
		this.sinal = valor >= 0 ? 1 : -1;
		this.quantia = new Dinheiro(moeda, 0, valor);
	}

	public ValorMonetario somar(Dinheiro quantiaSomada) {
		validarMoeda(quantiaSomada);
		Integer valor = quantia.obterQuantiaEmEscala() * sinal + quantiaSomada.obterQuantiaEmEscala();
		return new ValorMonetario(moeda, valor);
	}

	public ValorMonetario subtrair(Dinheiro quantiaSubtraida) {
		validarMoeda(quantiaSubtraida);
		Integer valor = quantia.obterQuantiaEmEscala() * sinal - quantiaSubtraida.obterQuantiaEmEscala();
		return new ValorMonetario(moeda, valor);
	}

	public Dinheiro obterQuantia() {
		return quantia;
	}

	public Boolean negativo() {
		return sinal < 0;
	}

	public String formatado() {
		return zero() ? formatarSemSinal() : formatarComSinal();
	}
```

```
42
43          public String formatarComSinal() {
44                  return negativo() ? formatarNegativo() : formatarPositivo();
45          }
46
47          public String formatarSemSinal() {
48                  return String.format("%s", quantia.formatado());
49          }
50
51          public Boolean zero() {
52                  return quantia.obterQuantiaEmEscala() == 0;
53          }
54
55          private String formatarPositivo() {
56                  return String.format("+%s", quantia.formatado());
57          }
58
59          private String formatarNegativo() {
60                  return String.format("-%s", quantia.formatado());
61          }
62
63          private void validarMoeda(Dinheiro quantiaSomada) {
64                  if (!moeda.equals(quantiaSomada.obterMoeda())) {
65                          throw new UnsupportedOperationException();
66                  }
67          }
68
69          @Override
70          public boolean equals(Object objeto) {
71                  if (objeto instanceof ValorMonetario) {
72                          ValorMonetario outro = (ValorMonetario) objeto;
73                          Boolean iguaisZero = zero() && outro.zero();
74                          Boolean iguaisComMesmaMoeda = sinal.equals(outro.sinal) && quantia.equals(outro.quantia) && moeda.equals(ou
75                          return iguaisZero || iguaisComMesmaMoeda;
76                  }
77                  return super.equals(objeto);
78          }
79
80          @Override
81          public String toString() {
82                  return formatado();
83          }
84
85  }
```

APÊNDICE II – ESTRATÉGIAS DE FIXTURE SETUP

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa1.inline;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;
import br.ufsc.ine.leb.sistemaBancario.SistemaBancario;

public class TesteBanco {

	@Test
	public void caixaEconomica() throws Exception {
		SistemaBancario sistemaBancario = new SistemaBancario();
		Banco caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
		assertEquals("Caixa Econômica", caixaEconomica.obterNome());
		assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa1.inline;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;
import br.ufsc.ine.leb.sistemaBancario.SistemaBancario;

public class TesteAgencia {

	@Test
	public void caixaEconomicaTrindade() throws Exception {
		SistemaBancario sistemaBancario = new SistemaBancario();
		Banco caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
		Agencia caixaEconomicaTrindade = caixaEconomica.criarAgencia("Trindade");
		assertEquals("001", caixaEconomicaTrindade.obterIdentificador());
		assertEquals("Trindade", caixaEconomicaTrindade.obterNome());
		assertEquals(caixaEconomica, caixaEconomicaTrindade.obterBanco());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa1.implicit;

import static org.junit.Assert.assertEquals;

import org.junit.Before;
import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;
import br.ufsc.ine.leb.sistemaBancario.SistemaBancario;

public class TesteBancoAgencia {

	private Banco caixaEconomica;

	@Before
	public void configurar() throws Exception {
		SistemaBancario sistemaBancario = new SistemaBancario();
		caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
	}

	@Test
	public void caixaEconomica() throws Exception {
		assertEquals("Caixa Econômica", caixaEconomica.obterNome());
		assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
	}

	@Test
	public void caixaEconomicaTrindade() throws Exception {
		Agencia caixaEconomicaTrindade = caixaEconomica.criarAgencia("Trindade");
		assertEquals("001", caixaEconomicaTrindade.obterIdentificador());
		assertEquals("Trindade", caixaEconomicaTrindade.obterNome());
		assertEquals(caixaEconomica, caixaEconomicaTrindade.obterBanco());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa1.delegate;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;

public class TesteBanco {

	@Test
	public void caixaEconomica() throws Exception {
		Banco caixaEconomica = Auxiliar.criarCaixaEconomica();
		assertEquals("Caixa Econômica", caixaEconomica.obterNome());
		assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa1.delegate;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;

public class TesteAgencia {

	@Test
	public void caixaEconomicaTrindade() throws Exception {
		Banco caixaEconomica = Auxiliar.criarCaixaEconomica();
		Agencia caixaEconomicaTrindade = Auxiliar.criarTrindade(caixaEconomica);
		assertEquals("001", caixaEconomicaTrindade.obterIdentificador());
		assertEquals("Trindade", caixaEconomicaTrindade.obterNome());
		assertEquals(caixaEconomica, caixaEconomicaTrindade.obterBanco());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa1.delegate;

import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;
import br.ufsc.ine.leb.sistemaBancario.SistemaBancario;

public class Auxiliar {

	public static Banco criarCaixaEconomica() {
		SistemaBancario sistemaBancario = new SistemaBancario();
		return sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
	}

	public static Agencia criarTrindade(Banco banco) {
		return banco.criarAgencia("Trindade");
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa1.dependency;

import static org.junit.Assert.assertEquals;

import org.junit.Before;
import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;
import br.ufsc.ine.leb.sistemaBancario.SistemaBancario;

public class TesteBanco {

	private Banco caixaEconomica;

	@Before
	public void configurar() throws Exception {
		SistemaBancario sistemaBancario = new SistemaBancario();
		caixaEconomica = sistemaBancario.criarBanco("Caixa Econômica", Moeda.BRL);
	}

	@Test
	public void caixaEconomica() throws Exception {
		assertEquals("Caixa Econômica", caixaEconomica.obterNome());
		assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa1.dependency;

import static org.junit.Assert.assertEquals;

import org.junit.Before;
import org.junit.Test;

import br.ufsc.ine.leb.projetos.estoria.Fixture;
import br.ufsc.ine.leb.projetos.estoria.FixtureSetup;
import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;

@FixtureSetup(TesteBanco.class)
public class TesteAgencia {

	@Fixture private Banco caixaEconomica;

	private Agencia caixaEconomicaTrindade;

	@Before
	public void configurar() throws Exception {
		caixaEconomicaTrindade = caixaEconomica.criarAgencia("Trindade");
	}

	@Test
	public void caixaEconomicaTrindade() throws Exception {
		assertEquals("001", caixaEconomicaTrindade.obterIdentificador());
		assertEquals("Trindade", caixaEconomicaTrindade.obterNome());
		assertEquals(caixaEconomica, caixaEconomicaTrindade.obterBanco());
	}

}
```

APÊNDICE III – TESTES INCOMPLETOS

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.inline;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Conta;
import br.ufsc.ine.leb.sistemaBancario.Moeda;
import br.ufsc.ine.leb.sistemaBancario.SistemaBancario;

public class TesteConta {

	@Test
	public void joaoCaixaEconomicaTrindade() throws Exception {
		assertEquals("0001-4", joaoCaixaEconomicaTrindade.obterIdentificador());
		assertEquals("João", joaoCaixaEconomicaTrindade.obterTitular());
		assertTrue(joaoCaixaEconomicaTrindade.calcularSaldo().zero());
		assertEquals(caixaEconomicaTrindade, joaoCaixaEconomicaTrindade.obterAgencia());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.inline;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;
import br.ufsc.ine.leb.sistemaBancario.SistemaBancario;

public class TesteAgencia {

	@Test
	public void caixaEconomicaTrindade() throws Exception {
		assertEquals("001", caixaEconomicaTrindade.obterIdentificador());
		assertEquals("Trindade", caixaEconomicaTrindade.obterNome());
		assertEquals(caixaEconomica, caixaEconomicaTrindade.obterBanco());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.inline;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;
import br.ufsc.ine.leb.sistemaBancario.SistemaBancario;

public class TesteBanco {

	@Test
	public void caixaEconomica() throws Exception {
		assertEquals("Caixa Econômica", caixaEconomica.obterNome());
		assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.implicit;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import org.junit.Before;
import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;

public class TesteBancoAgenciaConta {

	@Test
	public void caixaEconomica() throws Exception {
		assertEquals("Caixa Económica", caixaEconomica.obterNome());
		assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
	}

	@Test
	public void caixaEconomicaTrindade() throws Exception {
		assertEquals("001", caixaEconomicaTrindade.obterIdentificador());
		assertEquals("Trindade", caixaEconomicaTrindade.obterNome());
		assertEquals(caixaEconomica, caixaEconomicaTrindade.obterBanco());
	}

	@Test
	public void joaoCaixaEconomicaTrindade() throws Exception {
		assertEquals("0001-4", joaoCaixaEconomicaTrindade.obterIdentificador());
		assertEquals("João", joaoCaixaEconomicaTrindade.obterTitular());
		assertTrue(joaoCaixaEconomicaTrindade.calcularSaldo().zero());
		assertEquals(caixaEconomicaTrindade, joaoCaixaEconomicaTrindade.obterAgencia());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.delegate;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;

public class TesteBanco {

	@Test
	public void caixaEconomica() throws Exception {
		assertEquals("Caixa Económica", caixaEconomica.obterNome());
		assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.delegate;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.experimento.etapa1.delegate.Auxiliar;

public class TesteAgencia {

	@Test
	public void caixaEconomicaTrindade() throws Exception {
		assertEquals("001", caixaEconomicaTrindade.obterIdentificador());
		assertEquals("Trindade", caixaEconomicaTrindade.obterNome());
		assertEquals(caixaEconomica, caixaEconomicaTrindade.obterBanco());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.delegate;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Conta;

public class TesteConta {

	@Test
	public void joaoCaixaEconomicaTrindade() throws Exception {
		assertEquals("0001-4", joaoCaixaEconomicaTrindade.obterIdentificador());
		assertEquals("João", joaoCaixaEconomicaTrindade.obterTitular());
		assertTrue(joaoCaixaEconomicaTrindade.calcularSaldo().zero());
		assertEquals(caixaEconomicaTrindade, joaoCaixaEconomicaTrindade.obterAgencia());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.delegate;

public class Auxiliar {

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.dependency;

import static org.junit.Assert.assertEquals;

import org.junit.Before;
import org.junit.Test;

import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.Moeda;
import br.ufsc.ine.leb.sistemaBancario.SistemaBancario;

public class TesteBanco {

	@Test
	public void caixaEconomica() throws Exception {
		assertEquals("Caixa Econômica", caixaEconomica.obterNome());
		assertEquals(Moeda.BRL, caixaEconomica.obterMoeda());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.dependency;

import static org.junit.Assert.assertEquals;

import org.junit.Before;
import org.junit.Test;

import br.ufsc.ine.leb.projetos.estoria.Fixture;
import br.ufsc.ine.leb.projetos.estoria.FixtureSetup;
import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Banco;
import br.ufsc.ine.leb.sistemaBancario.experimento.etapa1.dependency.TesteBanco;

public class TesteAgencia {

	@Test
	public void caixaEconomicaTrindade() throws Exception {
		assertEquals("001", caixaEconomicaTrindade.obterIdentificador());
		assertEquals("Trindade", caixaEconomicaTrindade.obterNome());
		assertEquals(caixaEconomica, caixaEconomicaTrindade.obterBanco());
	}

}
```

```java
package br.ufsc.ine.leb.sistemaBancario.experimento.etapa2.dependency;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import org.junit.Before;
import org.junit.Test;

import br.ufsc.ine.leb.projetos.estoria.Fixture;
import br.ufsc.ine.leb.projetos.estoria.FixtureSetup;
import br.ufsc.ine.leb.sistemaBancario.Agencia;
import br.ufsc.ine.leb.sistemaBancario.Conta;

public class TesteConta {

	@Test
	public void joaoCaixaEconomicaTrindade() throws Exception {
		assertEquals("0001-4", joaoCaixaEconomicaTrindade.obterIdentificador());
		assertEquals("João", joaoCaixaEconomicaTrindade.obterTitular());
		assertTrue(joaoCaixaEconomicaTrindade.calcularSaldo().zero());
		assertEquals(caixaEconomicaTrindade, joaoCaixaEconomicaTrindade.obterAgencia());
	}

}
```

APÊNDICE IV – CASOS DE TESTE SISTEMA BANCÁRIO

Dado que:
Exista o sistema bancário.
Quando:
For criado o banco `Banco Do Brasil`.
Então:
O nome do banco será "Banco do Brasil".
A moeda do banco será BRL.

Dado que:
Exista o sistema bancário.
Exista o banco `Banco do Brasil`.
Quando:
For criada a agência `Centro`.
Então:
O identificador da agência será "001".
O nome da agência será "Centro".
O banco da agência será o `Banco do Brasil`.

Dado que:
Exista o sistema bancário.
Exista o banco `Banco do Brasil`.
Exista a agência `Centro`.
Quando:
For criada a conta `Maria`.
Então:
O identificador da conta será "0001-5".
O titular da conta será "Maria".
O saldo da conta será zero.
A agência da conta será `Centro`.

Dado que:
Exista o sistema bancário.
Exista o banco `Banco do Brasil`.
Exista a agência `Centro`.
Exista a conta `Maria`.
Quando:
For realizada a operação de deposito de dez reais na conta `Maria`.

Então:
A operação terá sido realizada com sucesso.
O saldo da conta `Maria` será de dez reais.

Dado que:
Exista o sistema bancário.
Exista o banco `Banco do Brasil`.
Exista a agência `Centro`.
Exista a conta `Maria`.
A conta `Maria` tenha um saldo de dez reais
Quando:
For realizada a operação de saque de seis reais da conta `Maria`.
Então:
A operação terá sido realizada com sucesso.
O saldo da conta `Maria` será de quatro reais.

Dado que:
Exista o sistema bancário.
Exista o banco `Banco do Brasil`.
Exista a agência `Centro`.
Exista a conta `Maria`.
A conta `Maria` tenha um saldo de quatro reais
Quando:
For realizada a operação de saque de seis reais da conta `Maria`.
Então:
A operação não terá sido realizada devido à saldo insuficiente.
O saldo da conta `Maria` será de quatro reais.

APÊNDICE V – ALTERAÇÃO SISTEMA BANCÁRIO

```
14      public Banco criarBanco(String nome, Moeda moeda, Dinheiro taxaPorOperacaoDeTransferencia) {
15              Banco banco = new Banco(nome, moeda, new Dinheiro(moeda, 0, 0));
16              bancos.add(banco);
17              return banco;
18      }
```

APÊNDICE VI – QUESTIONÁRIO DO EXPERIMENTO

1. Como você avalia sua experiência prática com desenvolvimento de sistemas?
    - Nenhuma, nunca tive experiência com desenvolvimento de sistemas antes.
    - Baixa, tenho pouca experiência com desenvolvimento de sistemas.
    - Média, tenho moderada experiência com o desenvolvimento de sistemas
    - Alta, tenho muita experiência com desenvolvimento de sistemas.
2. Como você avalia sua experiência prática com desenvolvimento de testes?
    - Nenhuma, nunca tive experiência com desenvolvimento de testes antes.
    - Baixa, tenho pouca experiência com desenvolvimento de testes.
    - Média, tenho moderada experiência com o desenvolvimento de testes.
    - Alta, tenho muita experiência com desenvolvimento de testes.
3. Quais estratégias de *fixture setup* você já conhecia (marque quantas for necessário)?
    - *Inline setup*.
    - *Implicit setup*.
    - *Delegate setup*.
4. Como você avalia o esforço para aprendizado da estratégia de *fixture setup* proposta:
    - Baixo, considero que é fácil compreender e aprender a usar a estratégia.
    - Médio, considero que o esforço para compreender e aprender a usar a estratégia é moderado.
    - Alto, considero que é difícil compreender e aprender a usar a estratégia.
5. Como você avalia o esforço para utilização da estratégia de *fixture setup* proposta:
    - Baixo, considero que é fácil usar a estratégia.
    - Médio, considero que o esforço para usar a estratégia é moderado.

    - Alto, considero é difícil usar a estratégia.
6. Como você avalia a utilidade da estratégia de *fixture setup* proposta:
    - Pouco útil, vejo que a estratégia tem utilidade prática em nenhuma ou quase nenhuma situação.
    - Útil, vejo que a estratégia tem utilidade prática em algumas situações.
    - Muito útil, vejo que a estratégia tem utilidade prática em todas ou quase todas situações.
7. Escreva outras considerações (se houver) a respeito da estratégia de *fixture setup* proposta.